DESIGNING SECURITY POLICIES AND FRAMEWORKS FOR WEB APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Kapil Singh

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2011

DESIGNING SECURITY POLICIES AND FRAMEWORKS FOR WEB APPLICATIONS

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Professor Nick Feamster
School of Computer Science
Georgia Institute of Technology

Professor Patrick Traynor
School of Computer Science
Georgia Institute of Technology

Dr. Mihai Christodorescu
IBM Research T. J. Watson

Date Approved: 20 May 2011

To my two wonderful families,

for providing endless love,

expecting nothing in return.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the help and support of a number of individuals who have influenced my life both professionally and personally. Even though no amount of gratitude would suffice for their kindly efforts, I would like to acknowledge some of the key contributors.

First of all, I want to express my sincere gratitude to my advisor, Wenke Lee, for his guidance and support, and for helping me build my ability to produce high-quality research. I am indebted to him for giving me the freedom and opportunity to work on research areas that were outside his core research focus, and for having continuous faith in me even during tough times in the PhD pursuit. I am also thankful for the excellent example he has provided as a successful researcher and professor.

I am also grateful to the rest of my committee members, Mustaque Ahamad, Nick Feamster, Patrick Traynor and Mihai Christodorescu for their invaluable feedback on my research that went a long way in improving the thesis. I would like to express my special thanks to Jonathon Giffin, Helen Wang, Alexander Moshchuk and Sumeer Bhola, whose support, collaboration and guidance was essential for my research.

I have had few of the most memorable years of my life at Georgia Tech and my labmates at GTISC have played the biggest role in providing a wonderful and friendly environment. I want to thank Abhinav, Andrea, Bryan, Chaitrali, Danesh, David, Guofei, Ikpeme, Junjie, Long, Manos, Martim, Monirul, Prahlad, Roberto, Vijay and everyone in the GTISC lab for their fruitful discussions, their advice, their research tips, and for being unselfishly supportive throughout my graduate life. I am proud of having been part of GTISC and I will cherish these memories for ever. My heart also goes with the College of Computing and its wonderful people. I made some great

friends here and they will always be a part of my life. I thank them all for their moral support.

I would like to thank my family for their endless love and support. They have always been an inspiration for me to dig deep and try hard to face the toughest challenges and to achieve the highest goals. One of the biggest influence in my life through graduate school has been my lovely wife, Claris, whose sacrifices, support, inspiration and love was absolutely essential for success of my PhD dissertation work. At times, our common big dreams in life helped fight and get through the most challenging moments in my PhD.

Finally, I offer my regards and blessings to all of those who I have not mentioned, but who have had a positive impact on me at different times of my life. I thank them for being part of the person that I am today, which has helped me produce this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

There are multiple players that participate in forming policies to determine the security of content on the Web. These players include web applications that determine who can access their content, users of these applications desiring control over security policies that determine sharing of their contributed content, and the client-side software such as web browsers that have mandatory enforcement of their security policies. The current web security policies do not satisfy the end-to-end security requirements imposed by this multi-player environment. For example, while average users desire control over security policies that determine sharing of their contributed content, the applications still control what access control policies are available to the users for controlling access to that content. Moreover, even if web security policies are improved to satisfy the new requirements, their enforcement still leaves much to be desired in the current web infrastructure. Existing mechanisms are ineffective in enforcing security and privacy policies in the evolving web environment thereby undermining the security of content on the Web.

In this dissertation, we explore ways to improve end-to-end security for web access by design and analysis of effective web security policies and enforcement frameworks. Our contributions cover end-to-end security solutions that are aligned with the multi-player setup of Web 2.0 and include a framework for users to specify security policies, a platform to enforce user policies for third-party applications, an analysis of browser policy issues, and a mechanism to provide improved end-to-end security/integrity guarantees.

We advocate the use of user-defined access control for the user-centric Web 2.0

environment and develop a generalized framework, called *xAccess*, for a user to specify policies on how data seekers can access the user's data in the context of web applications. xAccess is analogous to the single sign-on mechanism, however, instead of providing login capability, it provides the user with a single point for defining his access control models and policies for one or multiple applications. We subsequently extend our enforcement mechanism to develop a framework for application platforms to enforce user-defined policies with third-party applications, in particular to control flow of data. We use social networking as representative application and design a novel framework, called *xBook*, for building social networking platforms that uses information flow control models to enforce user's privacy policies.

We evaluate client-side security by performing a systematic analysis of the incoherencies in current browser security policies. Given that wide-scale adoption of any new browser policy, even if it is for improving security, is marked with concerns for backward compatibility, we also present the results of a large-scale compatibility study to analyze the cost of, and thus ultimately motivate, the adoption of secure browser policies.

Any meaningful security on the web browser platform cannot be ensured without achieving end-to-end security between a users web browser and web sites. Although HTTPS can help achieve end-to-end security by preventing man-in-the-middle attacks, it does not satisfy the requirements of web applications that desire improved performance at the cost of reduced security guarantees. To this end, we develop a new protocol, *HTTPi*, which offers only end-to-end authentication and integrity and seamlessly enable caching at intermediate servers (such as CDN servers and cache proxies). We subsequently propose mechanisms that allow web applications to place integrity policy requirements on the content embedded on their sites.

CHAPTER I

INTRODUCTION

1.1 Motivation

The advent of "Web 2.0" [66, 101] has changed the requirements for systems currently deployed on the Web, not only stretching the capabilities of the current infrastructure but also outstripping the current security and privacy protection mechanisms. Traditionally, web applications used to be the main providers of web content that was consumed by the end users. One of the biggest developments driving Web 2.0 is that the average users have become substantial contributors of web content, whether it is in the form of blogs, personal pictures or social interactions.

As a consequence of this change, multiple web participants (herein called *web players*) now drive the formation and enforcement of security policies that determine the end-to-end security of content on the Web. First, the web application hosted on a server determines who can access its content. Second, the client-side software such as web browsers have mandatory enforcement for their security policies. Finally, the average users contributing web content also desire more control over security policies that determine sharing of their content.

The trust relationships between these web players define the security requirements of the web content. While a user trusts his own web browser and the web application that he is accessing, he has no trust in other users of the application (note that users are unknown to each other by default). From the application's point of view, trust exists for the user's browser where the application is rendered, but no trust is placed in other web applications. The network is always considered untrusted by all the web players. Any end-to-end solution to security must effectively maintain

these trust relationships to have any security guarantees. However, the existing web infrastructure lacks the ability to effectively enforce security in line with the changing Web 2.0 requirements.

Traditionally, the web application's content was delivered explicitly to different users that did not involve any sharing of user content, therefore user-user trust relationship was correctly maintained. Since web application provided the content, the access control policies on this content was defined by the application. However, the access control mechanisms in current web applications do not align with the user-centric Web 2.0 setting: even if the content is provided by the users, the applications still control what access control policies are available to the users for controlling access to that content. As a result, users do not have much control over defining their own user-user trust relationships. To further stretch the security requirements, application development has become much more distributed with a growing number of users acting as developers for third-party applications. Web applications, in turn, are acting as programming platforms allowing developers to run third-party content on top of their frameworks. One such example is Facebook that became a platform in 2007. These third-part applications add another spectrum of trust relationship for the user who is contributing content, since these applications, by default, are not trusted by the user.

At the client side, web browsers have also gradually evolved from being an application that views static pages to a rich application platform to render dynamic content, such as maps. Techniques such as AJAX [48, 62] allow web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page. Other mechanisms such as client-side mashups [37, 136] allow multiple, mutually distrusting web site principals to co-exist and interact within the browser. Since Web 2.0 applications have more active interactions within the browser as compared to their earlier counterparts, it

creates new challenges for the browsers to satisfy the trust relationship between two distrusting web applications. However, current browser policies have been developed in an ad-hoc and piecemeal fashion to accommodate for the new features (such as mashups) and as a result, many trapholes exist in the current browsers that lead to security vulnerabilities and breach of trust for the user and the applications.

Even with sound mandatory browser security policies, the authenticity of the content provider and the integrity of its content are often at question since much of the Web is delivered over the *untrusted* network using HTTP rather than HTTPS. Consequently, network attackers can carry out man-in-the-middle attacks and undermine browsers access control, even when browsers flawlessly implement their access control mechanisms. The web applications are limited in their availability of options, with HTTPS providing security but suffering from performance overheads and lack of in-networking caching, and HTTP that supports in-networking caching with no security guarantees. With demand for improved user's experience growing, web applications often opt for performance over security by using HTTP and thus undermine the end-to-end security for web access.

1.2 Dissertation Overview

Through our analysis of new design goals and discovery of new security challenges introduced by Web 2.0, we suggest the following thesis statement:

> *Existing web security policies and enforcement frameworks do not satisfy the end-to-end security requirements of Web 2.0. Mechanisms that collectively consider the three web players–the user, the web browser and the application–to specify, evaluate and enforce security policies, can significantly improve end-to-end security for web access.*

Therefore, the focus of this dissertation is on how new tools and frameworks can be effectively designed to aid the protection of web systems by acting as policy

specification and enforcement points. This has culminated in four pieces of unique contributions.

First, we develop a generalized and application-independent framework, called *xAccess* [122], for a user to specify policies on how data seekers can access the user's data in the context of web applications. A simple example of such user-specific policies for a blogging application is that of a blogger defining different access to his blog for personal friends and office colleagues. Another user of the same application may only desire public or private access for his blog entries. Our framework provides an abstract base model to which the user's access control policies can be mapped: this single, unified abstraction allows the modeling of a wide range of access control policies specified by different users of an application. From a user's perpective, xAccess is analogous to the single sign-on mechanism, but instead of providing login capability, it provides the user with a single point for defining his access control models and policies for one or multiple applications. For a web application, it provides a single abstraction that can support multiple access control models and policies that are individually specified by its users. xAccess is a powerful tool that gives users freedom to change their access control model or policies at any time; such a change requires no changes in the underlying applications.

Second, we develop a framework for application platforms to support user-defined control for data sharing with third-party applications. One fitting example of such web applications is social networking that has recently transformed from being a service provider to a platform for running third party applications [80]. Users have typically trusted their platform application with personal data, and have assumed that their privacy preferences are correctly enforced. However, they are now being asked to trust each application they use in a similar manner. This has left the user information vulnerable to accidental or malicious leaks by these applications [67, 92]. We use social networking as representative application and design a novel framework,

called *xBook* [121], for building social networks that protect users' privacy in the presence of untrusted third-party site extensions. In contrast to user-user access control protection, privacy protection against third-party applications has additional challenges. Since untrusted running code is involved, our mechanism has to control not only what the third-party applications can access, but also what these applications can do with the data that they are allowed to access. For this purpose, we use information flow models to control what untrusted applications can do with the data they receive. We implement a proof-of-concept prototype of xBook, and evaluate its practicality by developing sample applications using its APIs.

Third, we conduct a systematic analysis of the incoherencies in current browser security policies [123]. One example of such policies is that current browsers support certain features that allow applications to have access to resources belonging to the user or to enable them to trick the user into perform unintended action. By uncovering such trapholes, we aim to enumerate all possibilities of data leaks from the browser and suggest policies to prevent these leaks. Given that wide-scale adoption of any new browser policy, even if it is for improving security, is marked with concerns for backward compatibility, we perform a large-scale compatibility study to analyze the cost of, and thus ultimately motivate, the adoption of secure browser policies.

Finally, meaningful security on the web browser platform cannot be ensured without achieving end-to-end security between a user's web browser and web sites. Although HTTPS can help achieve end-to-end security by preventing man-in-the-middle attacks, its universal adoption by web sites is hindered by its performance cost and its inability to be cached at intermediate servers (such as CDN servers and cache proxies). As our fourth contribution, we observe that only end-to-end authentication and integrity are required for the browser platform to enforce its access control reliably. Without end-to-end confidentiality, content can be cached. To this end, we propose a new protocol, *HTTPi* [124], which offers only end-to-end authentication

and integrity and seamlessly works with the existing web caching infrastructure. We also propose mechanisms that allow web applications to place integrity policy requirements on the content embedded on their sites. HTTPi performs content signing while perserving progressive content loading supported by browsers. Because content signing can be done offline, HTTPi incurs negligible overhead over HTTP. HTTPi fills the gap between HTTP and HTTPS effectively by providing the right level of security required by many web applications, while allowing the applications to achieve improved network performance.

1.3 Organization

The rest of this dissertation is organized as follows. We present the design of our user-centric xAccess framework in Chapter 2. We demonstrate the viability of our design by means of a platform prototype. We discuss the usability and performance of this framework, and also provide example applications to demonstrate how developers can utilize the framework.

Chapter 3 discusses the design and implementation of the xBook framework for building privacy-preserving social networking applications. We demonstrate how xBook uses information flow models to control what untrusted applications can do with the information they receive. The usability of the platform is evaluated by developing sample applications using the platform APIs. We also discuss both security and non-security challenges in designing and implementing such a framework.

Chapter 4 presents a systematic analysis of the current state of browser access control policies and describes several incoherencies in these policies that result in possible data leaks from the browsers. We subsequently suggest improved security policies to prevent such leaks.

Chapter 5 covers the design of the HTTPi protocol, which offers only end-to-end authentication and integrity and seamlessly works with the existing web caching

infrastructure. It also discusses a mechanism that allows web applications to specify and enforce their access control policies. We analyze the security and performance of the protocol and discuss new access control policies associated with HTTPi.

Finally, we discuss related work in Chapter 6 followed by conclusions in Chapter 7.

CHAPTER II

USER-CENTRIC WEB APPLICATION POLICIES

With the advent of Web 2.0 technologies, web application development has become much more distributed with a growing number of users acting as developers and sources of online content. In particular, many users are contributing more and more contents, by providing their personal information on social networks or by adding information in the form of blogs, reviews, etc.

While the trend is towards more user-contributed data, the mechanisms to define the access control policies on user-contributed data are still under the control of the web applications. Consider the example of social networks such as Facebook. Users contribute data in the form of their profile information, by loading pictures, or by posting messages in each other's profiles. The mechanism to control access to the users' data is determined by the social networking web site; in most cases, it is limited to a small number of pre-defined access categories such as private, public or to providing access only to the users' friends. As a result, the users are forced to use alternate means to protect their privacy in today's applications, for example, by maintaining multiple blogs [85]. Moreover, users have to understand and subsequently remember the access control policies and customized rules for each web application, which can be a cumbersome task considering the number of applications used by a typical user.

Even if a web site or application wants to change its access control mechanism to satisfy the needs of its users, there are major obstacles. First, the diversity in the user population and the variety of the data contributed by each user means that developing a mechanism that caters to the need of every user might not be feasible.

Second, even though the privacy expectations that users desire are easy to state, there is still a large gap between users' mental models and the policy languages of the current access control systems provided by the applications [36].

We argue that the users are the ones best suited to decide the semantics and importance of their own data. Thus, we need to provide users the freedom and utility to define the access control policies specific to their data, and enable web applications to enforce these potentially diverse policies.

In this work, we propose a unified framework for providing access control in web applications that provides users with a wide range of access control options to satisfy their own individual privacy requirements and is independent of any application. The framework is unified in the sense that it provides the user with a single point for defining his access control models and policies for one or multiple applications. This is akin to a single sign-on mechanism [18] that enables specification of access control instead of providing login capability. In our design, the users have the flexibility to define their own policies to control the privacy of the data contributed by them. A simple example of such user-specific policies for a blogging application is that of a blogger defining different access to his blog for personal friends and office colleagues. Another user of the same application may only desire public or private access for his blog entries. Our framework provides an abstract base model to which the user's access control policies can be mapped: this single, unified abstraction allows the modeling of a wide range of access control policies specified by different users of an application. Note that the goal of this work is *not* to propose a new access control model, but to develop a unified system that gives users an ability to define customized access control policies for all their applications at a single point.

We demonstrate the viability of our design by implementing a prototype system, called xAccess. xAccess has two components, one that runs on the client side as an extension to the user's browser and another component that is hosted on the server

of the web application. A user desiring access control for his data defines his policies using an interface provided by the xAccess extension. The extension translates these user policies to categories in our base access control model. Our model is based on the Role Based Access Control (RBAC) scheme, so these categories translate to specific roles in the model. xAccess also provides an interface to apply the user-defined policies to any granularity of data desired by the owner, for example, to protect individual blog entries, particular personal information, or specific photos. Furthermore, xAccess also allows specific words or phrases within a blog to be tagged with user-defined categories, thereby allowing data owners to control access to specific information, such as someone's name in a blog entry. Only a user who wants to control access to his data is required to install the xAccess' browser extension; the extension is not needed by any other user who is seeking access to the owner's data. For the rest of the chapter, we use the term *owner* to denote an individual providing data and *seeker* to denote a person who wants access to the owner's data.

The server-side component of xAccess uses the access categories of the base model to determine whether a particular access should be granted. Our model not only controls access to the read operation, but also supports access control to other operations like write or download. In a blogging application, the server-side component filters a blog considering the categories attached to various parts of the blog. By default, a reader is only presented with the public entries of an owner's blog. Further access is granted only after the owner's policy specifies a category label for the reader and the access is limited to that category. In a wikipedia application, the owner can similarly restrict write operation to his wiki entry by attaching appropriate write permission to the entry.

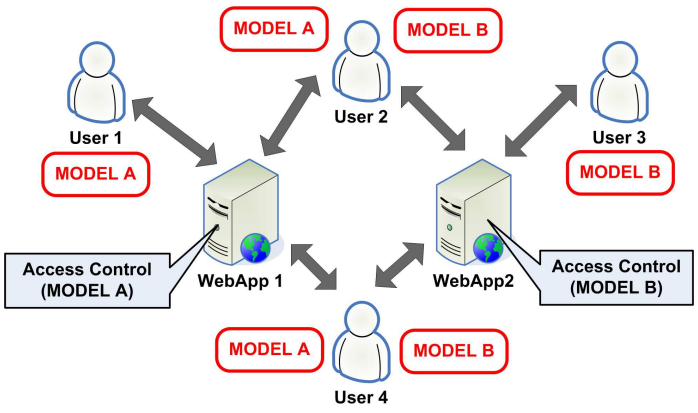
One of the strengths of xAccess's abstract model is that it allows other access control models to be incorporated into our framework. This is a desirable feature,

because we expect other current and future access control models to facilitate development of more diverse user policies closer to an owner's mental model. Our base model is generic and can simulate a wide range of such models (Section 2.2.2). For example, in the blogging application, new models such as Content Based Access Control (CBAC) [69] can support policies like "only people mentioned in the blog should see the blog". Our framework supports such models, thereby supporting more diverse user policies, without requiring any change to our base model and without any modifications to the web application.

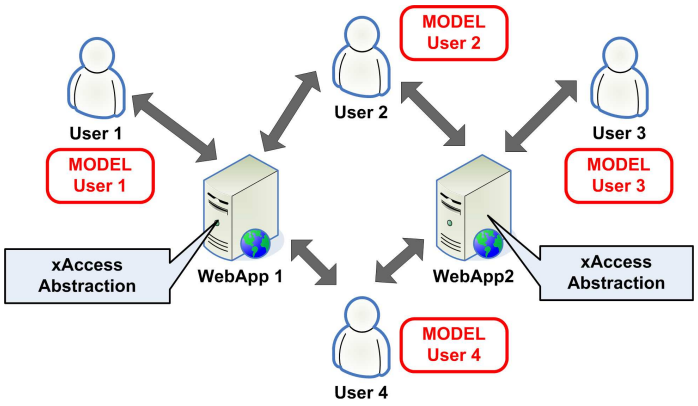
2.0.1 Limitations of Current Designs

The current access control design of web applications enforces an application-specific single security policy for every user of that application. However, there are different degrees of intimacy between an individual and any other user desiring access to his data, but the current designs are too coarse to capture these distinctions. In most current access control systems, the owner of data has no say in defining the granularity of access allowed.

These coarse-grained policies might result in undesired access or undesired restrictions. For example, users on Flickr only have the option to assign their office colleagues to the category of family, friends, or public. Assigning them public access would prevent them from viewing some of the "professional" pictures posted by the user. On the other hand, giving them friends access would allow them to see any pictures accessible to friends, which might not be desirable for some users. On the flip side, some users might not want to share some of the professional pictures with a few or all of their friends or family due to confidentiality constraints. The web application is ill-equipped to decide the access granularity required by a particular owner; only the owner is familiar with his own particular situation and relationships



(a)



(b)

Figure 1: Application architecture for: (a) current frameworks. (b) xAccess framework.

to provide the right level of access to other users in the system. Most web applications currently employ limited number of access control categories with little or no flexibility in adding new user-specific categories.

The web application is also restricted in terms of the access control mechanism it uses. In the current setting, most applications provide a single access control model for all users of the application (Figure 1(a)). For example, Facebook uses access control lists to evaluate access to a user's profile, photos, messages, etc. This mechanism is common for all the users, independent of individual requirements. Additionally, the mechanism is more-or-less fixed and there is no incentive for web applications to change the model unless there is a huge user base vouching for that change. As a

result, new innovative mechanisms such as CBAC [38] might take years before being adapted by the web applications. Even if some applications are more proactive to the change, it does not hold true for most web applications.

2.0.2 Our Contributions

Our framework design allows an owner to keep a single access control model that can be utilized for one or multiple web applications (Figure 1(b)). However, the owner also has the option to use different models for different applications. For example, the framework empowers the owner to deploy access control mechanisms like MAC [115] or CBAC [69] allowing him flexibility in writing his own access control policies. From any web application's prospective including the ones that currently do not employ access control, one-time installation of our framework enables the application to support different access control models for different owners.

Our work makes the following contributions:

- We propose a novel design of an unified access control framework for supporting diverse user-defined access control policies that empowers the users to choose their own models and their own access granularity. We also show that our model is *generic* to allow simulation of a number of popular access control models on top of our framework, and provides enormous *flexibility* to the users in making access control decisions about the data owned by them.

- We develop a proof-of-concept system, called xAccess (extended Access), that provides a set of APIs that can be used to integrate generalized access control capability into web applications.

- We demonstrate the viability of our framework by developing a sample blogging application as our base example and subsequently integrating access control into the application using the xAccess APIs. We also show real-world deployment

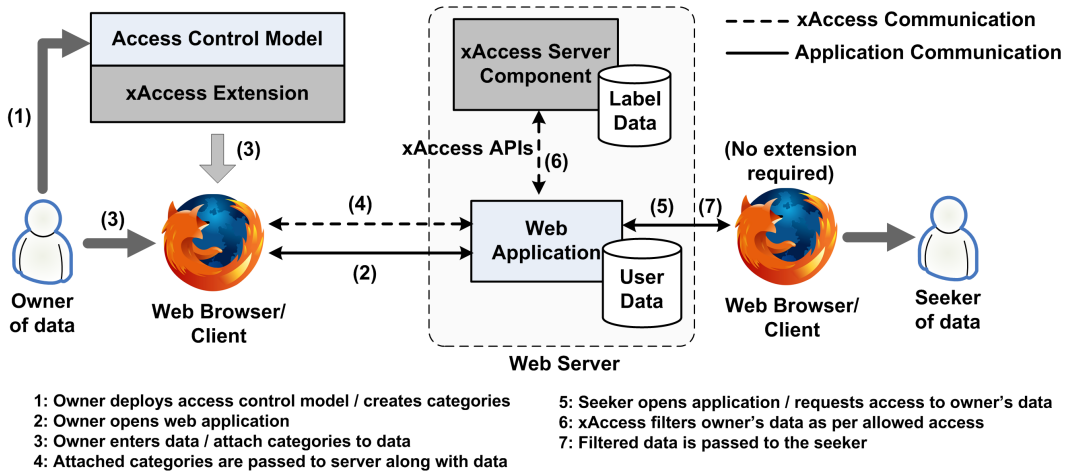


Figure 2: `xAccess` Architecture and workflow scenarios.

potential of our framework by integrating xAccess into a popular open-source wikipedia application. Our sample web applications are available online [16,23].

Chapter Organization. The rest of the chapter is organized as follows. We present an overview of the xAccess framework in Section 2.1. We present the base model for access control in xAccess in Section 2.2. Section 2.3 presents the implementation details and evaluation of xAccess. We discuss the advantages and limitations of xAccess in Section 2.4, followed by conclusions in Section 2.5.

2.1 The `xAccess` *Framework*

xAccess is a framework for enabling web applications to capture and model data owners' privacy policies and to enforce such policies via access control on data seekers. xAccess is designed to be general and adaptive so that it can be used for a wide variety of web application scenarios and more importantly for different access control models implied by owners' diverse policies. More specifically, xAccess only requires one-time installation at the server side; after this initial installation, no change is required at the server and owners are free to change their policies via a client side component at any time. Furthermore, xAccess provides a generalized access control model, based on the Role Based Access Control (RBAC) model [56,116], to represent policies specified

either in the form of user-defined access classifications (such as private, friends, family, etc.) or in the form of other access control models (such as MAC [115], DAC [114], CBAC [69], etc).

Figure 2 shows a high-level design of the xAccess framework. There are two parts of the xAccess platform, one that is hosted on the server-side and the other that runs on the client-side as an extension to the data owner's web browser.

On the client side, the xAccess extension provides a layer of abstraction that enables its RBAC-based generalized access control model (which we call *base model*) to directly represent policies specified in terms of user-defined access categories as well as convert policies specified using other access control models. As we will show in Section 2.2.2, widely used access control models, including Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC), can be constructed on top of the base model in xAccess. These models are simulated in xAccess by setting various parameters of the base model using the APIs provided by xAccess.

xAccess provides multiple avenues for a data owner to express his access control requirements. First, it provides the owner with a capability to define access categories directly into the base model via xAccess' browser extension. Second, it enables the owner to extend xAccess' base model by deploying other access control models of his choice, such as CBAC, as a layer above xAccess. When deployed, such a model is coupled with xAccess into the browser extension allowing the owner to write his access control policies using the interface (or language) provided by the deployed model (Figure 2). We expect such model transformations to be made available by third-party developers providing the end users with easy-to-use interfaces to specify their access control policies, while keeping them oblivious to the implicit transformation to xAccess' base model. In comparison to the end users, we assume that such developers would be better equipped to correctly develop such interfaces. The user interface

provided by xAccess forms part of the browser's chrome and does not modify an application's code. It supports access control for both structured information (such as well-defined fields in a user's personal profile) and unstructured data (such as blogs). The xAccess extension only needs to be installed at the owner's web browser, and is not required for any other user only seeking access to the owner's data.

On the server side, the xAccess component receives and stores the mapping of entities (both subjects and objects) with their corresponding categories that are passed from the client to the application's server. The xAccess component is a separate module on the server side; it is only invoked by the web application when required to filter content that is access controlled (Figure 2). The amount of modifications required to integrate xAccess into an application is negligible (3–5 lines of API calls).

The enforcement of access control is done by the server side component of xAccess, which serves all users of the application and in effect realizes their corresponding access control models. Since the translation of user-specific access control models to xAccess' base model is done at the client side and only the access categories corresponding to the base model are presented to the server, this greatly simplifies the access control enforcement at the server side. xAccess' server-side component uses a simple matching algorithm to decide if a requesting subject (seeker) is allowed to access an owner's data object. The algorithm uses categories of the subject and the object in making such a decision. We present details of this algorithm when we introduce our model in Section 2.2.1.

In our current design, the server side of xAccess is invoked as a set of API calls made by the web application. An alternate design would be to deploy xAccess as a proxy that filters data before passing it to the user's browser. The difference from our current design is subtle and a detailed comparison is not the focus of our work.

2.2 User-Centric Access Control

In this section, we present the generic access control model that underlines our xAccess framework. We call this the base model. It has the following design goals:

- **Generalization.** Since one of the goals of the xAccess framework is to allow data owners to specify their access policies, which can be very diverse, the base model should be able to accommodate a wide range of access control models. That is, it should allow policies defined using different access control models to be expressed or simulated in the base model. It should also model policies expressed in terms of access categories, as commonly used in web application scenarios.

- **Minimum modification requirements.** A web application should only require one-time, minimal modifications at the time of deployment of xAccess. Once the application deploys the xAccess framework, it should require no further changes to the application even if a data owner changes his access model at any later time.

- **Backward compatibility.** Even after deploying the xAccess framework, an application should still support users with no xAccess component installed on the client-side. In other words, the fall-back mechanism of xAccess should be the same (default) behavior of the application when no xAccess is deployed.

In order to achieve these goals, we designed xAccess to be policy neutral. The base model in xAccess provides an abstraction of the essential elements of any access control policy, which would at the minimum include the access categories or role hierarchies, and the constraints and administration of user-role and role-permission assignments. We next describe the base model, and show how it enables diverse user policies to be modeled and enforced in the xAccess framework.

2.2.1 The Base Model

We constructed our model by customizing the Role Based Access Control (RBAC) model for our *user-centric* paradigm. The central idea of RBAC is that permissions are associated with roles and users are made members of appropriate roles thereby acquiring the roles' permissions. Roles are created for the various job functions in an organization, and users are assigned roles according to their responsibilities and qualifications. The roles are assigned by the system administrator of the organization. RBAC allows for the specification and enforcement of a variety of protection policies, which can be tailored on an enterprise-by-enterprise basis. The RBAC framework provides administrators with the capability to regulate who can perform what actions, when, from where and in what order.

The goal of the xAccess framework is to provide protection to user data in accordance with the access control defined by the data owner. In this regard, user's contributed data (i.e., user profile, blogs, photos, etc.) for a particular web application represents a habitat that corresponds to an organization in the RBAC model. In our model, we associate the administrator privileges for the access control over any data items to the owner of those items. The data owner defines the roles for his "system" because he is the one who knows the "responsibilities and qualifications" of various individuals, acting as users of the application, from his personal connections to people. For example, a data owner knows who his family members are in real life and how much each person can be trusted with his data. In other words, a role in our model signifies a real-life relationship of the data owner. This relationship could be in the form of family, friends, business colleagues, public or any others role specific to the data owner. Such roles might vary from user to user. There is many-to-many mapping from roles to users; this also shows similarity to the the real world, where an individual might have multiple friends or family members, or a friend could also be part of the family thereby assuming both roles.

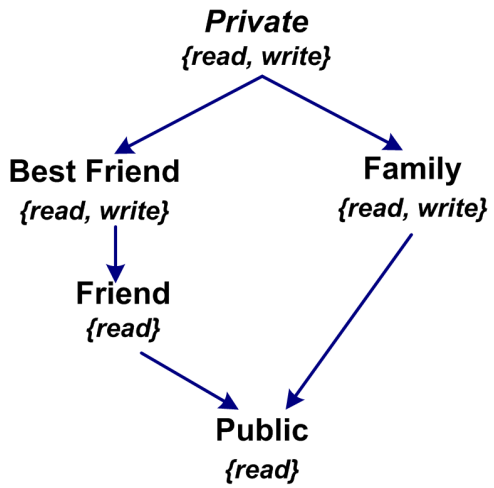


Figure 3: Example role hierarchy in `xAccess`. Text in *italics* represents corresponding permissions.

In RBAC, a role signifies a set of operations that can be performed by that role. In our model, these operations can take limited forms depending on the application. In most cases, the operation would be limited to data read, allowing a data owner to control the confidentiality of his data. Some typical applications include social networks and blogs. In other cases, the data owner might want to assign write permissions to the data contributed by him. Wikipedia is one application that falls into this category. Providing access control for other operations such as direct download, remote execution, etc. is also feasible in our system, if such operations are supported by the application.

The xAccess framework tracks and enforces access control using a labeling system defined based on the existing RBAC models [56, 116]. Subjects represent the users requesting data access and objects represents the data entities that are being requested. Both subjects and objects correspond to roles in the xAccess framework.

Subjects are associated with roles when the request for access is granted by the web application on behalf of the data owner. The objects are assigned roles when data is entered into the application by the data owner, e.g., by uploading new photos, writing new blog entry, adding new profile information, etc. A data owner can update

roles of both subjects and objects anytime at his own discretion.

Role Hierarchy. Roles can have overlapping responsibilities and privileges, that is, users belonging to different roles may need to perform common operations on some objects. For example, a best friend should be able to read the data accessible to a friend's role. To efficiently satisfy such requirements, our model includes specification of role hierarchies. An example of a role hierarchy within xAccess is shown in Figure 3. In this example, the role `Private` "contains" the roles of `Best Friend` and `Family`. This means that members of the role `Private` are implicitly associated with the operations, constraints, and objects of the roles `Best Friend` and `Family` without the administrator having to explicitly list their attributes for the `Private` role. The most powerful roles are represented at the top of the diagram with the less powerful roles being represented at the bottom. That is, the roles on the top of the diagram contain the greatest number of operations, constraints, and objects. As shown in Figure 3, not all roles have to be related. The roles `Friend` and `Family` are not hierarchically related but they can contain some or all of the same roles.

xAccess allows new constraints to be added into the model at any stage. Constraints are set of rules that are mandated over any role assignments and hierarchy definitions. They define a broad scope of what is acceptable in the xAccess model. For example, a user may want to enforce a rule that none of his office colleagues can be on his family list. This ensures that information about his family activities or family pictures are kept hidden from his office colleagues. Accordingly, a constrain can be added using the xAccess APIs that enforces that no user can be assigned the twin roles of `Family` and `Office Colleague` at the same time.

Note that xAccess enforces no restriction on what roles can be defined. It only provides a framework that can be utilized to define roles and role hierarchies that can effectively simulate the access control model underlying the user-defined access

Algorithm 1 Access Control Algorithm of xAccess.

```
INPUT S: subject, O: object, REQ_OP: operation requested
if application supports REQ_OP on O then
  R_s ← getRoles(S)
  R_o ← getRoles(O)
  for all r ∈ R_s do
    for all r' ∈ R_o do
      if r ⪰ r' and REQ_OP ∈ getOperations(r) then
        GRANT S with REQ_OP access to O
        return
      end if
    end for
  end for
end if
DENY S with REQ_OP access to O
```

Figure 4: Pseudo code of the algorithm to check if the Subject S can perform operation REQ_OP on Object O. REQ_OP can be in the form of read or write on any other application-specific operation. *getRoles(x)* returns the roles corresponding to the entity x and *getOperations(y)* returns the permissions for the role y. $r \succeq r'$ represents the role hierarchy where role r is higher than (or contains) role r'.

control policies. Section 2.2.2 will discuss how our base model can simulate the commonly used access control models, thereby enabling a generic access control system for diverse user and application needs.

The server-side framework of xAccess stores the roles for both subjects and objects associated with each user, the hierarchy of roles, as well as the corresponding allowed operations such as read or write for each role. It also enforces the access control rules on behalf of the data owner. Figure 4 shows a simplistic view of the algorithm used in the xAccess framework to evaluate whether or not to grant access to the seeker for the requested data item or resource. As can be observed, a subject can perform a specific operation (read, write, etc.) on an object only if the operation is supported by the application and allowed by the access control rules of the object's owner. lThe simplicity of this algorithm demonstrates the value of our design where a single algorithm is able to effectively cover a wide range of access control models

used by different data owners.

2.2.2 Expressiveness of the Base Model

One of the requirements of the xAccess framework is that it should allow users (or third-party developers acting on behalf of the users) to choose their own access control models. In order to fulfill this requirement, the xAccess base model should be generic enough to capture the functionality of the chosen access control model. In other words, the abstractions in the base model should be expressive enough to enforce a wide range of access control models.

For most current web pages, access control is enforced by just customizing the access levels or categories and by providing partial order to such categories. For example, in Facebook such categories are friends, friends of friends, private, etc. These simple policies can be directly modeled by the base model and the default interface provided by xAccess. However, we anticipate that future web applications may require more elaborate access control policies.

In this section, we leverage previous research [70,102,114] to demonstrate that our RBAC-based model can be used to simulate few traditional and most commonly used models: Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Lattice Based Access Control (LBAC) [51, 115]–and their variants. We supplement these findings by additional discussion on how some newer models such as Content Based Access Control (CBAC) [69] are realized within xAccess. A more formal discussion of such simulation is out of scope of this thesis and is left as future work.

2.2.2.1 DAC

Discretionary Access Control (DAC) is a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission on to any other subject. Sandhu et al. demonstrated that

several variations of DAC can be simulated via the RBAC model. The basic idea behind the DAC to RBAC construction is to simulate the owner-centric policies using roles that are associated with each object [114]. In the xAccess framework, the owners have discretion to transfer certain controls (read, write or execute) to other users. In our default setup, transfer of controls is allowed only if the original owner provides ownership permission to other users. Depending on his own requirements, the owner can create additional permissions to restrict transfer to only specific controls.

2.2.2.2 MAC / LBAC

Mandatory Access Control (MAC) refers to a system of access control that assigns security labels or classifications to objects and allows access only to subjects with distinct levels of authorization or clearance. In contrast to DAC, where a subject can pass permissions to access an object to other subjects, no such propagation is allowed in the MAC model. These controls are enforced by the security administrator of the base system.

Lattice Based Access Control (LBAC) is a special type of MAC where a lattice is used to define the levels of security that an object may have and that a subject may have access to. A subject is only allowed to access an object if the security level of the subject is greater than or equal to that of the object. For example, in the xAccess system, a user can give different levels of access to top friends, friends or family. Based on how the lattice is constructed, LBAC can be used for confidentiality, integrity, or both.

Osborn et al. have demonstrated that the LBAC model can be simulated using RBAC [102]. Since our base model is a customized form of RBAC, any variations of the MAC model can be transformed to our base model. In our model, an owner acts as the administrator for his own data representing the system administrator in the MAC model.

2.2.2.3 CBAC

Content Based Access Control (CBAC) is a type of access control in which access to an object is partially or entirely based on the content of the objects in the system [38, 113]. CBAC allows users to specify a single, intuitive access control policy based on object features and then automatically applies that policy to new objects as they are created. CBAC utilizes techniques from natural language processing [113], image processing [38] and machine learning to perform this task. Essentially, it provides the necessary bridge between a user's intuitive access control policy, such as "Parents should not see my party pictures", and the policy enforcement.

CBAC is currently an open research project, with efforts to improve its practical acceptability to a variety of applications [69]. The advantage of CBAC is that it reduces the burden of managing the access control policies for the users – the users just need to provide high-level policies and CBAC controls the policy enforcement with no further intervention required from the user.

The advantage of our xAccess framework is that it allows CBAC to be integrated into a web application easily using our abstractions. It would also enable easy integration of CBAC into the application at any time in the future, i.e., whenever users think that it is viable enough for their purposes. Note that xAccess places no guarantee or control over the correctness of CBAC; the users are empowered to decide whether or not to use the CBAC systems. An alternative to our approach in the current systems is that the applications themselves switch to the CBAC model. However, applications have to cater to the need of all its users to consider a new technology and hence might take much longer to switch to the new CBAC model. Furthermore, switching to a new model again limits all users of the application to that model.

To deploy CBAC on top of our xAccess framework, the implementation of CBAC needs to convert high-level user policies in their model to lower level roles, subjects and objects in the xAccess model. For example, let us consider a simple CBAC policy

"Only my friends can see my party pictures". First, the CBAC implementation uses image processing to determine which pictures are from "parties" and tag all such pictures with a "friend" role. The user could have approved a list of individuals to be added in the friend role. More advanced implementations of CBAC can analyze the user's data to infer the friend's list of the user based on who talks with whom.

Since the CBAC policies can be effectively implemented using RBAC, we can infer that the CBAC model can be simulated using the xAccess system.

2.2.3 Access Control Lifecycle in xAccess

We can now summarize the process by which any access control model defined by a data owner by means of the client (browser) component of the xAccess framework is mapped to the enforcement of access control at the server.

1. In the absence of any access control model specified by the user, xAccess uses its own base model by default. The xAccess browser extension presents a user interface to the data owner for defining his access preferences directly into our model. This interface allows the data owner to create roles and the corresponding role hierarchy. Alternatively, the data owner may choose to deploy his own access control model over xAccess (Figure 2). Such models implement their access control logic by defining roles and roles hierarchies using the APIs provided by xAccess' browser extension. Orthogonally, the implementation of these models can provide their own interface for the owner to define the model-specific policies. We believe that user-friendly interfaces can be developed by third parties to allow easy configuration of access control policies, which is both intuitive and easy to understand for the users; the usability aspect of such interfaces is beyond the scope of this work.

2. For each data entity – both structured and unstructured – input by a data

owner to an xAccess-compatible web application, the data owner has the option to attach a category. Such categories are pre-defined based on the access control model specified in Step 1. For the default base model, these categories correspond to the set of defined roles. If any other access control model is used, any category attached to a subject or an object is transformed to its corresponding role in the base xAccess model. For example, the CBAC model allows user policies like "My parents should not see my party pictures". The CBAC system has the capability to determine if a particular picture is a party picture based on the its pixel contents [69]. A typical CBAC interface provided to a picture's owner would allow him to upload the picture to the web application. The CBAC system accordingly tags the picture by passing a role mapped to the "party picture" category to the xAccess extension. Finally, the mapping of the entities with their corresponding roles is passed to the application's server to be stored by the server-side component of xAccess.

3. Any user of a particular web application can view the default public information (i.e., public profile, public blog, etc.) of all other users. However, in order to access non-public information, the seeker sends a request for access to the owner of that information. This logic is internal to the application and is similar in most typical applications.

4. A data owner sees all the access requests from other users, and associates categories to each requesting user according to the access control model and the categories chosen by him in Step 1. The assignment of roles to the requesting users is conveyed to the server-side of xAccess. Again, this procedure is aligned with most existing applications where a user needs approval from a profile's owner before accessing the profile.

5. The requesting seeker can now access the data of another user (the data owner)

in accordance with the roles assigned to him. The access control enforcement is done by the xAccess component on the server side using the algorithm given in Figure 4.

6. A data owner can modify the category of any of the objects contributed by him and any of the subjects approved by him, effectively changing the access allowed to these subjects.

2.3 Evaluation

2.3.1 Prototype System and Applications

We developed a working prototype of the xAccess system, which includes the server-side platform code and APIs for integrating the access control framework into the web applications. We also implemented the labeling model that provides the required abstraction at the server so that any access control model implemented at the browser side for the data owner can be enforced. Our xAccess platform consists of about 2500 lines of JavaScript code.

We demonstrate the viability of our approach by means of two sample applications. First, we developed a blogging application in-house that stores the profile information for users and provide them the functionality to write their personal blogs. Second, we use a popular open-source wikipedia application, called mediawiki, to show the applicability of our system directly to existing applications. By default, these applications have open access with any user logged into the application being able to view the profile and data (blogs/wikis) contributed by other users. In order to show the feasibility of our approach, we integrated the xAccess platform into the server side of these applications. This integration represents one-time installation of our xAccess framework for any application supporting the generic access control models. We also showed that such an integration incurs minimal changes to the existing code of the applications; the change comprises of few xAccess API calls to filter the data being

Table 1: Set of xAccess APIs for integrating the generic access control model.

`filterString(seekerID, ownerID, string)`	filters the *string* text according to the access control model of *ownerID* to give assess to *seekerID*
`isAccessAllowed(seekerID, ownerID, op)`	verifies if *seekerID* is allowed to perform the operation *op* on *ownerID*'s data
`setSubjectRole(ownerID, userID, role)`	assigns the *role* for *userID* in *ownerID*'s list
`setObjectRole(ownerID, objectID, role)`	assigns the *role* for *objectID* in *ownerID*'s list
`setRoleHierarchy(ownerID, roleHierarchy)`	updates the role hierarchy for *ownerID* on the web server

passed to the user.

On the browser side, we developed a sample proof-of-concept access control model and user interface that allows a user to design his own customized hierarchy of access control. The sample model is developed as a browser extension and has been tested for Firefox 3.5. Using the extension, users can perform addition, deletion or modification of new categories of access and customize the hierarchy graph of these defined categories. Our proof-of-concept model uses the xAccess APIs (given in Table 1) to map these user-defined categories to the abstractions used at the server side. The example blogging application and the wikipedia application integrated with our xAccess framework can be accessed online at [23] and [16] respectively.

To handle unstructured data such as blogs and wikis, our extension allows users to attach a category by selecting text on their browser window. This enables the user to select complete or part of the blog or the wiki. A new identifier tag is attached to the selected text and the corresponding identifier to category mapping is stored.

Figure 5 shows a partial overview of our server-side implementation by means of our blogging application example. In this example, a user *user1* has three subjects, namely *alice*, *bob* and *charlie*, which the user has assigned categories of `Friend`, `Family` and `Private`, respectively. He has defined the role hierarchy for his system

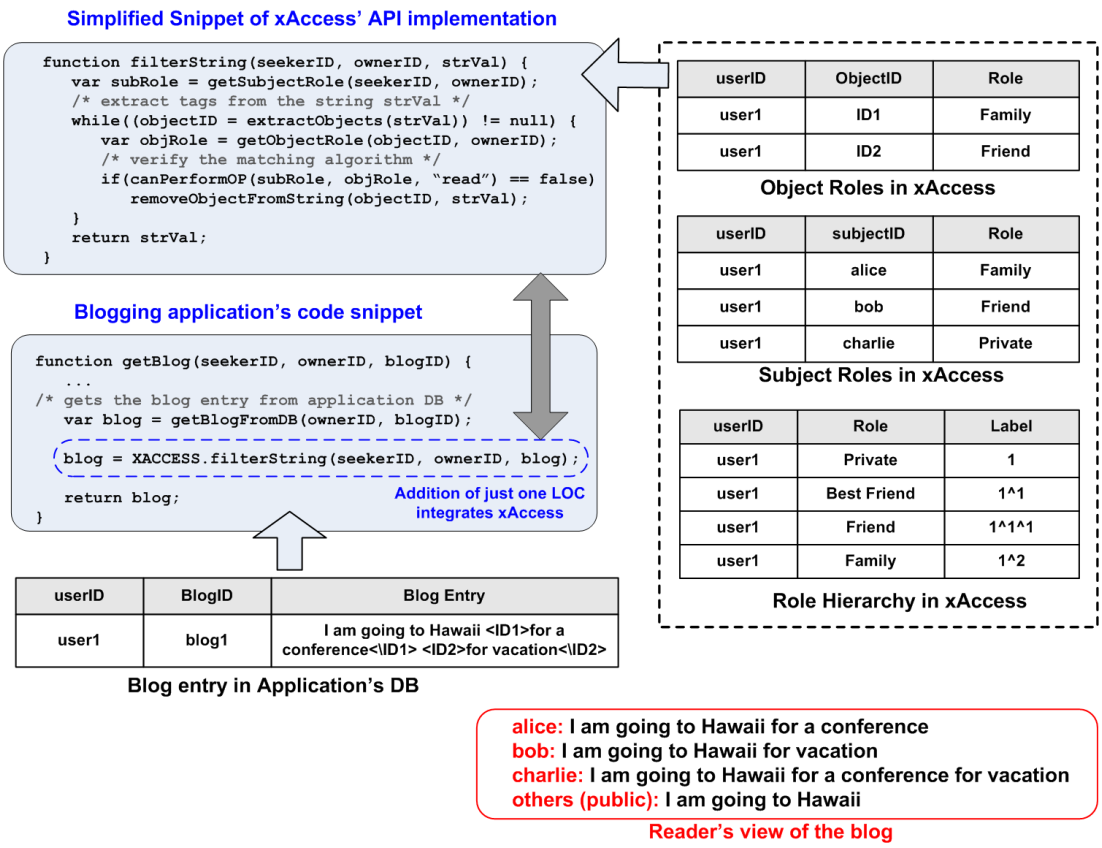


Figure 5: Blog example with sample xAccess API implementation.

as given in Figure 3. xAccess stores numbered representation for the role hierarchy in its database. Taking an example, role `Private` (represented as 1) has two children `Best Friend` and `Family`, labeled as $1 \wedge 1$ and $1 \wedge 2$, respectively. By separately storing the role hierarchy, we allow modification of the hierarchy without requiring any changes to the roles already assigned to the subjects and the objects.

Let us assume that the user makes a blog entry saying "I am going to Hawaii for a conference for vacation" and wants to provide different purpose of his travel to friends and family. The user tags parts of the blog with different categories to achieve his purpose using the default interface provided by the xAccess extension. The modified text is stored in the application's database and the corresponding categories in xAccess objects' database table. By tagging the text with identifiers instead of the actual roles, xAccess facilitates access modifications on the text objects without

changing the actual text stored with the application.

For any user accessing this blog, the server side of the application invokes the `filterString()` API of the xAccess platform (Table 1) before passing the returned value to the requesting user. Note that this requires addition of only one line to the application code. `filterString()` filters the blog entry by invoking xAccess' access control algorithm according to the roles assigned to the individual text objects and the requesting subject. As we can see from Figure 5, different readers have their own views of the blog based on their assigned roles. While *alice* sees the entry as "I am going to Hawaii for a conference", *charlie* can view the whole blog entry as his `Private` role is higher in hierarchy to both tags `Friend` and `Family` attached to different parts of the entry.

We use similar text tagging method to control access for structured data that are text fields, even though the interface provided to the user is different. For non-text fields such as photos, the categories are assigned to the filenames. The web application uses the `isAccessAllowed()` API to verify if access is allowed, before passing these entities to the requesting seeker.

2.3.2 Performance Estimates

xAccess does not impose a substantial burden on the performance of the web applications. Without being able to deploy the framework to a real-world application setup, it is difficult to accurately predict the impact of our design on the performance of these applications as perceived by the users. To get a rough estimate of the cost of supporting the xAccess design and the overhead involved in our system, we conducted some experiments with our sample applications, measuring the latency introduced by added security provided by xAccess' access control mechanisms.

In our experiments, the xAccess server is hosted on a 2.4GHz Intel Quad Core 2 machine with 4GB of RAM. The requests are made from Firefox 3.5 browser on

Table 2: User latency of various operations in typical web applications with xAccess.

Application Component	Operation	Number of access checks	Latency	
			Mean	Std Dev
User Profile (structured)	Read	12	37.1ms	0.83ms
Blog Entries (unstructured)	Read	5	16.9ms	0.60ms
Wiki Entries (unstructured)	Read	5	1.8ms	0.18ms
Wiki Entries (unstructured)	Write	5	5.8ms	0.46ms

a 2.33GHz, 2GB RAM, Pentium Core Duo laptop. Each test was run 10 times and measurement values were averaged. We define user latency as the difference in the time when the request is made at the browser and the time at which the response is received by the browser. Table 2 shows the latency introduced for various user interactions for the sample applications. Each interaction performs a different amount of processing based on the number of access control checks made for the interaction. For example, the number of checks required for the user profile is fixed at 12, one each for every profile field. On the other hand, the filtering of the blog (or the wiki) depends on the number of different access control tags added by the user in the unstructured blog (or wiki) entry. In our tests, each of the sample blog and the wiki entry had 5 such access checks. Our results show that the user latency for applications employing xAccess for providing access control is still considerably low: when averaged over the number of access checks, the latency is about 3.1ms for structured user profile fields and 3.4ms for the unstructured blog. For the open-source wikipedia application, providing access control for the unstructured wiki incurs an average latency value of 0.4ms for read and 1.2ms for write operation.

Studies have shown that acceptable user latencies fall in the range of 50–150 ms [119]. All user latencies observed in our experiments currently fall within this range. However, the latency increases with the number of checks added by the user. We emphasize that our prototype implementation is written in JavaScript with no

emphasis on optimization. There are many opportunities for improving the performance in our system, by optimizing the database queries or by utilizing server-side caching. Moreover, on a cluster of commercial servers with much better computational capacity, these values will be even smaller. Although it is not possible to precisely determine the cost of our approach without a large scale experiment, both the details of our design and the results from these experiments, support the conclusion that xAccess design imposes additional latency within acceptable limits.

2.4 Discussion

The support for both structured data (e.g., user's profile) and unstructured data (e.g., blogs) improves the ability of our framework to be acceptable in more diverse applications and environments. As previous research has suggested [64], ability to apply user-defined policies to a more finer grained level, such as words or phrases in blogs, satisfies a key access control requirement in the Web 2.0 paradigm. Other potential applications of our framework include email communication and newsgroups where a sender is passing messages to multiple receivers, either to anonymous groups or to unmanageably huge lists. Our framework allows the sender to attach desired access control tags to different parts of a message without explicitly identifying each receiver and creating separate individual messages. The filtering in turn is done by the sender's email server.

It might be argued that our design of pushing the access control models to the user's client might limit the mobility of the data owner, i.e., the ability to retrieve his access control preferences from the browser of any machine. However, the solution to this limitation could be trivial: since the web application is already storing the roles and role hierarchy for the user, it can allow downloading of such preferences to the user's client after login. Moreover, this is only required if the user wants to use a new machine to modify his access control preferences. In case no such modifications

are desired by the user, he can normally browse the web application with no need for xAccess' browser extension.

While our abstraction model of access control can simulate a wide range of access control models and, in consequence, the variations of these models [42, 86, 87, 141], there is still a possibility that our design might restrict some other models that cannot be mapped directly to the abstractions presented by our framework. While xAccess' design relies on the abstraction provided by the base model, it has the flexibility to accommodate an improved abstract model. This work is a first attempt to design a unified, single-point, user-centric access control framework and can certainly benefit from further research in the field of access control.

2.5 Summary

We presented a generic access control framework, called xAccess, that allows users to control how they want their data to be accessed. On one hand, xAccess is generic enough to enable users to choose their own access categories and on the other hand, it also supports integration of other access control models to further increase the diversity of policies available to the users. Our framework allows users to utilize a single unified access control across multiple web applications. From an application's prospective, it enables the application to support different access control models deployed by its users using a single model abstraction.

We developed a working prototype of the system and showed its viability by integrating two sample applications with our framework using the xAccess APIs. The blogging application was developed in-house and represents a typical web application with need for providing access control to its users [23]. We also integrated our framework into a popular open-source wikipedia application [16].

Our system shows promise in supporting potentially valuable future access control models that could be targeted by individual users to control access to their data. Since

no change is required to deploy a new model after the one-time installation of our framework, any future models chosen by the users can be easily integrated to any web application.

CHAPTER III

PRIVACY PRESERVING DESIGN FOR WEB APPLICATION PLATFORMS

In the last chapter, we presented a framework to enforce user-defined access control policies for user data contributed to trusted applications. This chapter extends our enforcement solution to include untrusted third-party applications. We use social networking as our representative example for our solution; similar design is applicable to other web application platforms.

Social networking sites have transformed the way people express themselves on the Internet and have become a door to the social life of many individuals. Users are contributing more and more content to these sites in order to express themselves as part of their profiles and to contribute to their social circles online. While this builds up the online identity for the user, it also leaves the data vulnerable to be misused, as an example, for targeted advertising and sale.

More private data online has lead to growing privacy concerns for the users, and some have faced extreme repercussions for sharing their private information on these networking sites. For example, students have been fined for their online social behavior [103]; a mayor was forced to resign because of a controversial Myspace picture [118]. There are numerous such cases, and these incidents clearly underline the importance of privacy control in social networks.

With the advent of Web 2.0 technologies, web application development has become much more distributed with a growing number of users acting as developers and source of online content. This trend has also influenced social networks that now act as platforms allowing developers to run third-party content on top of their

framework. Facebook opened up for third-party application development by releasing its development APIs in May 2007 [80]. Since the release of the Facebook platform, several other sites have joined the trend by supporting Google's OpenSocial [19], a cross-site social network development platform.

These third-party applications further escalate the privacy concerns as user data is shared with these applications. Typically, there is no or minimal control over what user information these applications are allowed to access. In most cases, these applications are hosted on third party servers that are difficult to monitor. As a result, it is not feasible to police the data being leaked from the application *after the data is shared with the application*. There have been several reported cases where users' private information was leaked by the applications, either due to intentional leaks [67] or due to vulnerabilities in the application [92].

Most social networking platforms, such as Facebook, currently provide the applications with full access to user profile information. This permission is granted in Facebook when the user adds the application, which requires the user to make a trust decision. Setting fine-grained access control policies for an application, even if they were supported, would be a complex task. Furthermore, access control policies are not sufficient in enforcing the privacy of an individual: once an application is permitted by a user's access control policy, it has possession of the user's data and can freely leak this information anytime for personal gains. For example, a popular Facebook application, Compare Friends, that promised users' privacy in exchange for opinions on their friends later started selling this information [132].

In this work, we are concerned with protecting the users' private information from leaks by third-party applications. We present a mechanism that controls not only what the third-party applications can access, but also what these applications can do with the data that they are allowed to access. We propose and implement a new framework called `xBook` that provides a hosting service to the applications

and enforces information flow control within the framework. xBook provides three types of enforcement that encapsulate the privacy requirements in a typical social network setting: (1) user-user access control (e.g., access to only friends) for data flowing within one application, (2) information sharing outside xBook with external parties; and (3) protection of the application's proprietary data. While (1) and (2) protects the privacy of a user from information leaks, (3) prevents the application's proprietary data or algorithm from being leaked to the application users.

The third-party applications are redesigned in such a way that they have access to all the data they require (allowing them to perform their functionality) and at the same time, not allowing these applications to pass this data to an external entity unless it is approved by the user. *Our framework enforces that the applications make these communications explicit to the user* so that he is more informed before approving an application.

There are several challenges associated with the design of our xBook framework:

Confinement. The execution of application code needs to be confined. This problem needs to be dealt with independently on the client side within the browser and on the server side in the web server. We use "the web server" as a conceptual entity to represent one or more servers.

Mediation. All communication from and within an application needs to be mediated by the xBook platform for permissible information flow. To this end, we developed a labeling model that enforces user-defined security policies. High-level policies specified by the user are converted to low-level labels enforced by xBook.

Programmability. The programming abstraction to the application writers should be practical and easy to use. xBook provides a set of simple APIs in line with the existing social networking platforms.

Portability. The requirements imposed by xBook on the application design should not break the existing applications. In other words, it should be feasible to port most

functionality of typical applications to xBook with little effort.

We show the viability of our framework design by implementing a working prototype of our xBook system and porting some of the popular applications from existing social networks, such as Facebook, on top of the framework. We also demonstrate a practical deployment strategy of our system by porting our framework itself as an application on Facebook. We evaluate the security of our platform by illustrating some possible application scenarios, and how xBook ensures privacy control in such cases. We also create some synthetic attacks that attempt to exploit the platform to leak information. Our results illustrate that xBook can successfully prevent all such attacks. Our performance results further demonstrate that xBook's privacy control mechanism incurs negligible overhead for typical social networking applications.

Chapter Organization. The rest of the chapter is organized as follows. Section 2 motivates our work by analyzing some privacy issues with the current social networking platforms. We present an overview of our xBook framework in Section 3. Section 4 and 5 discuss the implementation details of xBook's client-side and server-side components, respectively. Our labeling model is described in Section 6. Section 7 presents the evaluation results. We discuss the limitations of our work in Section 8. Finally, Section 10 provides a summary of the work.

3.1 Background

3.1.1 Social Networking Platforms

Social networks are the backbone of the online social life of many Internet users. These networks have expanded their development scope by allowing third-party developers to write their own applications, which in turn can be accessed and executed via the social network. An application is an entity that provides some value-added service to the user, and it requires user's profile data to perform its functionality. For example, a simple horoscope application generates daily horoscope based on user's

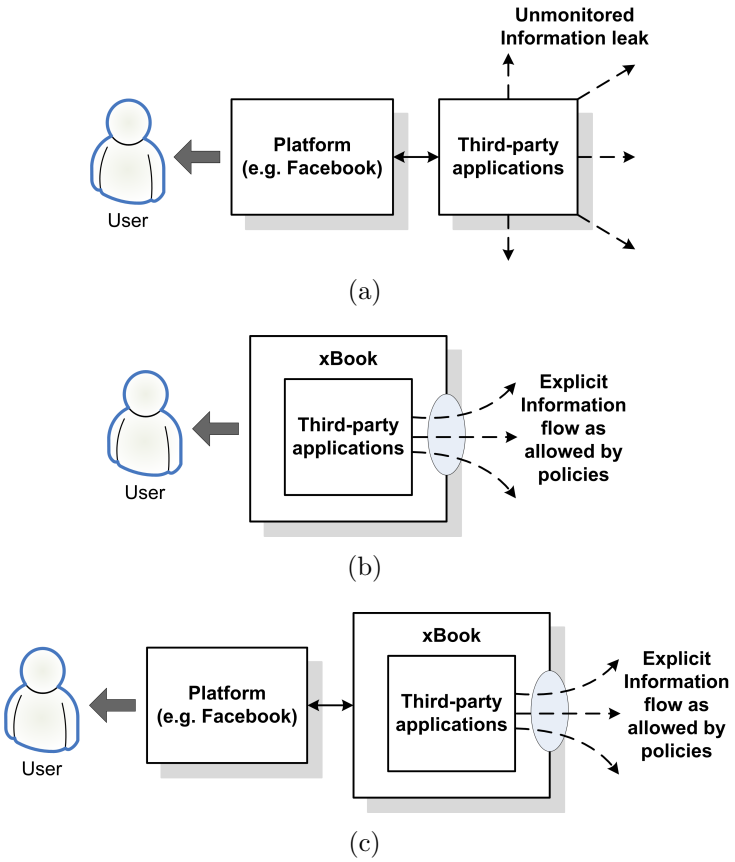


Figure 6: Application architecture for: (a) current platforms. (b) xBook platform. (c) xBook on Facebook.

birth information.

Facebook is one popular network that has pioneered the concept of the social network as a platform. The applications bring value both to the platform and its users in providing new features. Applications are deployed on their own servers and Facebook only acts as a proxy for integrating the applications' output to its own pages. The growing popularity of applications on Facebook has enticed other networks, such as Google's Orkut, to start supporting applications. The Orkut platform model is based on the OpenSocial framework [20]. OpenSocial provides a set of APIs for its partner sites (which it refers to as "containers") to implement. An application that is built for one container should be able to run with few modifications on other partner sites. The APIs allow third parties to have access to the social graph and personal

user data.

For the rest of the chapter, we use the Facebook case as an example; similar concepts apply to other social networking platforms.

3.1.2 Privacy Issues with Current Designs

Facebook supports customized policies for user-user access control, but currently provides no control on what user profile data can be accessed by third party applications. Applications run on their own servers that have no control administered by Facebook (Figure 6(a)). Applications need data to perform their functionality; they can request user data from the social platform and store it at their own servers. Facebook discourages storing user data on the application's own servers by barring it in their license agreement [10], but there is no way of enforcing it in Facebook's current architecture.

Application developers have access to a user's data even when they are not friends with the user. Unlike a regular friend relationship, this relationship is neither symmetric nor transparent: the application developer has access to the user's information, but the user does not necessarily know who the application developer is.

Before adding an application, the users are required to agree to a service agreement that allows the application to have access to their profile data. This general agreement is presented for every application, and no other specific information is provided about the application. Since a majority of the applications are known not to exploit users' personal data, the users tend to add any application, effectively defeating the purpose behind the service agreement. Additionally, second-degree permissions that allow applications to have access to the profiles of the users' friends add another layer of complexity.

There have been several reported incidents where users' information was leaked due to a vulnerability in the application [92]. The platform is trusting all third party developers, but the trust is misplaced since there is no restriction on who is allowed to

develop an application. One of the most popular Facebook applications, TopFriends, had a vulnerability that allowed any user of TopFriends to see the profile of another user, even if they are not friends with each other [92]. Private information of some high profile users was leaked. Facebook's response to this controversy was that they *expect* third party applications to follow their policies, which is not acceptable considering that there is no effective way to police the application developers.

User data has a lot of commercial value to marketing companies, competing networking sites, and identity thieves. Therefore, it is not surprising that many applications have been observed to intentionally leak user data to external parties for profit [67]. Other surveys have also discovered similar violations based on an application's externally-visible behavior [55]. The situation could be even worse as it is not feasible to determine how many other applications violate the user's privacy with internal data collection.

Social networking sites have a responsibility to protect user data that has been entrusted to them. The current approach is to legally bind the third parties using a Terms of Service (TOS) agreement [9]. However, it is not possible to monitor the path of information once the information has been released to these parties. Therefore, social networks can not rely on untrusted third parties following their TOS agreements to protect user privacy. Instead, privacy policies should be enforced by the platform and applied to all data that has been entrusted to the social networking site. Our platform design, `xBook`, is one step forward in this direction.

Felt et al. [55] have proposed a solution to proxy the user information in the form of tags to the third-party applications. These applications do not have access to user data and instead use pre-defined tags to format their output being displayed to the user. Their solution limits the capability of some important and popular applications, such as the horoscope application, that perform processing on user data beyond just displaying it. Our work enforces no such restriction on the application behavior.

3.2 xBook *Overview*

xBook is an architectural framework for building social networks that prevents untrusted third-party applications from leaking users' private information. The applications are hosted on xBook's trusted platform (Figure 6(b)), and xBook provides complete mediation for all communication to and from these applications.

In a social network setting, an application might communicate with entities outside the xBook system, called *external entities*, to perform specific tasks. For example, the horoscope application may communicate with `www.tarot.com` to receive horoscopes for every sunsign. The application also encapsulates its own data or algorithm that needs to be protected from untrusted users.

In the xBook framework, applications are designed as a set of *components*; a component being the smallest granularity of application code monitored by xBook. A component is chosen based on what information the component has access to and what external entity it is allowed to communicate with. In the horoscope application, one component communicates with `www.tarot.com` and has no access to user data. Another component has access to user's birthday, but does not communicate with any external entity.

From an end user's perspective, the applications are monolithic as the user does not know about the components. At the time of adding a particular application, the user is presented with a manifest that states what user profile data is needed by the application and which external entity will it be sharing this data with. For example, horoscope's manifest would specify that it does not share any information with any external entity. Note that the horoscope application does not need to reveal that it communicates with `www.tarot.com` as no user information is being sent to `www.tarot.com`. The user can now make a more informed decision before adding the application. Admittedly, the user will need to make a trust decision with respect to the parties with which the application shares user data, but these external parties

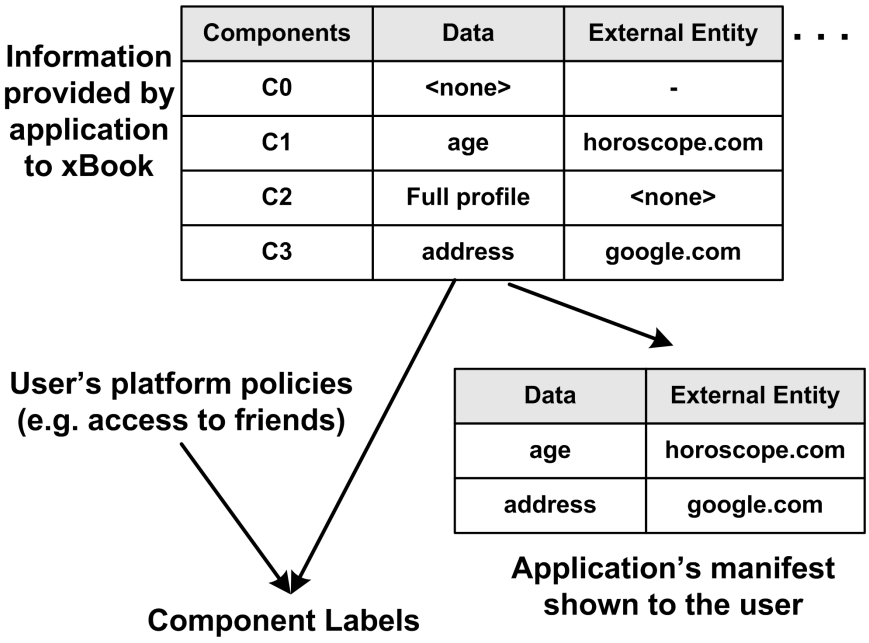


Figure 7: Typical life cycle of an application in xBook.

can be expected to be larger and better branded entities providing internet services, such as Google for ads, Yahoo for maps, etc.

Figure 7 shows a typical life cycle of an application. The developer of an application decides on the structure of the components for that application and during the application's deployment on xBook, he specifies the information required by each component and the external entity a particular component needs to communicate with. xBook uses this information to generate the manifest for the application. As shown in the figure, a manifest is basically a set that specifies all of the application's external communications (irrespective of the components) along with the user's profile data that is shared for each communication. Additionally, the xBook platform ensures that all of the application's components comply with the user's privacy policy and the manifest approved by the user. We discuss this further using the case study of an example application in Section 3.5.3.

The division of an application into multiple components allows the application writer to develop different functionality within an application that rely on different

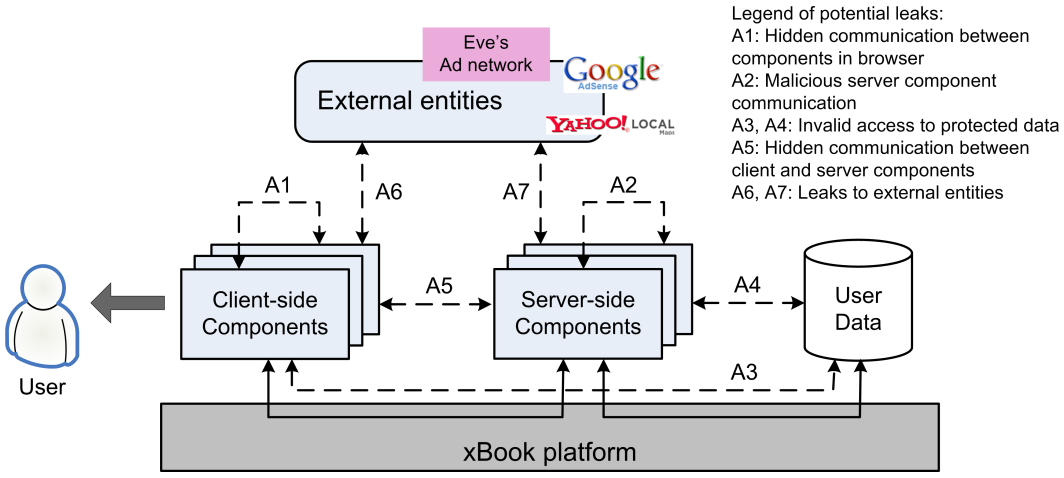


Figure 8: `xBook` architecture shown along with sources of potential leaks.

pieces of the user profile. For example, let us consider an application that requires a user's information to generate a customized profile for the user. It also requires his address information to be passed to Google to generate a map showing the address. In the application design of current social networks, the application would be able to pass all information about the user to Google. In the xBook framework, the application would be split into two components: the first component presents the customized profile of the user, has full access to the user's data and is not allowed to communicate with Google; the second component encapsulates the user's address (with no mapping to the user's profile) that is passed to Google to generate the map. We discuss some example applications in Section 3.6.1.

Figure 8 shows a high-level design of our xBook framework. There are two parts of the xBook platform, one that runs on the server-side and another that executes on the client-side in the user's browser (Figure 8). The application components, in turn, are also split into client-side and server-side components. The components are written in a safe subset of JavaScript, called ADsafe [2], which facilitates confinement of these components in our xBook implementation. Any communication to and from the components occurs by using xBook APIs, thereby allowing all such communication to be mediated by xBook. Each component is associated with a privilege level or

label that is derived from the application's manifest. The platform mediates the information flow between the components based on these labels (Section 3.5).

Both client-side and server-side components communicate with server-side storage to retrieve data. There are two types of storage in xBook system: one for storing xBook data that includes user profiles, and second for the data stored by the application. While the structure of xBook data is known, the semantic of the application data is internal to the application and hence unknown to the platform. All data fields are labeled to control access by application components. These labels are assigned based on high-level user-defined policies, such as a policy allowing access to only the user's friends, and the manifest approved by the user (Figure 7).

To store application data with unknown structure and semantics, xBook contains a group of storage pools, where data is stored as a set of name-value pairs. An application can have multiple storage pools, which could be for each user or for user-independent data.

3.2.1 Leakage Prevention by xBook Design

In the current platform designs, a user's information can be leaked in three major ways: (1) applications can share user's information with any third party, including advertisers, or fraudulent parties [67], and as shown in Figure 6(a), there is no way such a leak can be monitored in the current designs; (2) an application can pass information of one application user to another user, breaking free from the platform restriction that only friends can view a user's profile. The reported vulnerability in TopFriends allowed such a leak [92]; (3) the application can recreate the social graph of all its users by connecting common friends as edges in the graph.

xBook's design enforces complete mediation of all communication with the external entities (Figure 6(b)), thus preventing these applications from leaking information, effectively preventing (1) by design. A separate application instance is created for

every user, and that instance only has a view of the data accessible to that user. Data access is restricted to allowed user policies, such as access to friends. We mediate any direct or indirect communication between the components of two application instances, thereby deterring (2). (3) is prevented as no single component of an application can have direct access to the data of all its users: a component can only access an anonymized view of this data set (Section 3.4.2).

xBook, by design, solves most of the leakage problems of the current platforms. However, there are still some potential mechanisms to leak information in our system. We enumerate these possible threats in our formal model and address these threats one by one throughout the chapter.

3.2.2 Formal Requirements

We present a formal model in this section that generalizes xBook's mediation of untrusted third party applications. We use this model to analyze possible attacks, in terms of potential data leaks, under an adversary that deploys an application for collecting users' private data. We also identify a list of requirements that our system should satisfy in order to defeat such attacks. These formal requirements drive the design and architecture of our system.

Consider an application A consisting of a set of client-side components and a set of server-side components. Let U be the set of all users of the platform and Y be the set of all external entities. Suppose the application is allowed to communicate to a set of external entities $X \subseteq Y$ and a set of users $F_u \subseteq U$ for a particular user $u \in U$ who is using the system. Now, we divide the set of all data items D into three categories. First, there is a set of proprietary data or code of the application represented as $d_A \subseteq D$. Second, the set of data items $d_{u \to x}$ belonging to the user $u \in U$ that the application can transfer to the external entity $x \in X$. This set could be in the form of user's age, interests, photos, etc. Third, for an application instance

of user $u_i \in U$, the set of data items $d_{u_i \to u_j}$ is what the application can transfer to a user $u_j \in F_{u_i}$.

The platform wants to monitor the occurrence of a set of events E that can pass information outside an application component. Any event $e \in E$ is actively monitored by intercepting the information flow path between the point of the event occurring and the point where the event is handled. The platform monitors the content information I_e contained in the event. We express the response of the platform when the particular instance of the event has potential leaking information as $R(I_e)$, which may include filtering the content, blocking the communication, etc.

We can identify several sources of potential leaks in the xBook system (Figure 8). The first class of attacks (A1) bypasses the active monitoring by the xBook platform to leak private information from one client-side component to another, by creating a prohibited flow. Such attacks exploit some of the abstract features of the development language and the browser to leak information maliciously. In other words, A1 occurs if response $R(I_e)$ is not triggered even if the I_e contains private information content that is being leaked. Similar leaks (A2) are possible on the server-side where application components can break out of the sandbox to create a prohibited channel with other components. In addition, some attacks (A3 and A4) can occur during a component's access to data store, where the component gains access to restricted user or application data. Leaks (A5) can also occur in the communication between client-side and server-side components. Other attacks (A6 and A7) leak private information to entities outside the system. The leaks could be to an $x \in Y$ that is prohibited ($x \notin X$), or it could be leaking restricted piece of information $d \in D$ to an entity via communication that is allowed by the system, i.e., for $x \in X, d \notin d_{u \to x}$ for a user $u \in U$.

We completely forbid cross-application communication, effectively preventing leaks across applications. We also prevent direct communication between server-side components, only allowing them to communicate via storage, thereby preventing attacks

of type A2. We mediate other communication paths based on the labels of the communicating parties (Section 3.5). We address all other identified classes of attacks in Section 3.6.3. The requirements of an ideal social networking platform that guides the xBook design are as follows:

- Response $R(I_e)$ is invoked if I_e contains prohibited private information. In other words, the platform should be able to monitor any event that might be potentially leaking information, and should take action to prevent such leaks.

- Applications can invoke an event e iff $e \in E$, i.e., applications are restricted to a limited set of events for passing information to external entities.

- Application component having access to user u's private data d can send information to an external entity $x \in Y$ iff $x \in X$ and $d \in d_{u \to x}$. In other words, the platform should enforce user policies by limiting the communication to only *allowed* external parties and passing only *allowed* information to these parties.

- Application component having access to user u_i's private data d can send information to another component acting for user u_j iff $u_j \in F_{u_i}$ and $d \in d_{u_i \to u_j}$. This means that the applications should inherit the user-user access control policies of the platform.

- Application component x can access d_A only if $x \in S$, i.e., only server-side component of the application should have access to application's proprietary data.

We do not cover attacks against the browser in this work and assume that the browser behaves non-maliciously. Although phishing attacks can entice the user in choosing policies that might leak user information, we do not consider such attacks here. This work enforces the policies specified by the user, and does not consider social engineering attacks against the user.

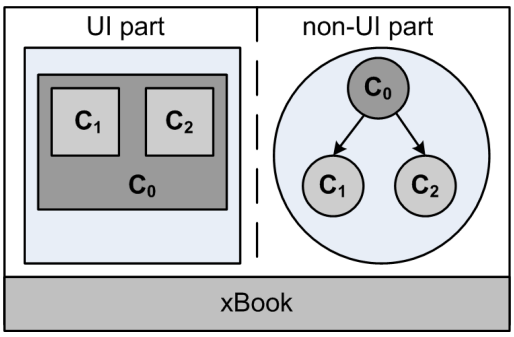


Figure 9: Client-side components in `xBook` design. C_0, C_1 and C_3 correspond to various components of a sample application.

3.3 Client-side Components

The client-side of the xBook platform and the client components of the applications run within the web browser. The components are further divided into two parts: the user interface (UI) part that is visible as part of the page to the user, and the non-UI part that provides communication interfaces with the external parties and with the server side. There is a one-to-one mapping between the non-UI and the UI parts, i.e., for every non-UI part, there is a corresponding UI part visible to the user (Figure 9).

A component is allowed to create another component. Information can flow during the component creation and this opens up the possibility of an information leak. We prevent such leaks by allowing components to create other components that are at least as restricted as the creating component. This principle prevents the creating component from leaking information out of the system via a less restrictive component.

At the front end, the creating component needs to delegate some screen space to the created component. One challenge is to isolate the third-party application components within the Document Object Model (DOM) of the webpage. A DOM is a platform- and language-independent standard model for representing HTML or XML documents in a browser. We present our confinement approach in the next section.

3.3.1 Confinement Mechanism

The components of an application encapsulate different levels of private information for the users. Therefore, these components need to be isolated from each other in order to prevent information leaks. On the client side, the components form a part of the DOM of the web page. The web page's DOM may include multiple components from one or multiple applications, apart from the platform's DOM objects.

In the current browser specifications, any script in a page has intimate access to all of the information and relationships of the page. As a result, the components are free to access information about the DOM objects of other components. In order to confine the components within their own control domain, we limit the application code to be written in an object capability language called ADsafe [2]. In an object capability language, references are represented by capabilities and objects are accessed using these references. Other alternatives to ADsafe, such as Caja [91], are also available; we decided in favor of ADsafe due to its simpler design and easier feature addition and customization to meet our system needs.

ADsafe. ADsafe defines a subset of JavaScript that makes it safe to put guest code (such as third-party scripted advertising or widgets) on any web page. ADsafe removes features from JavaScript that are unsafe or grant uncontrolled access to browser elements. Some of the features that are removed from JavaScript are global variables and functions such as `this`, `eval` and `prototype`. It is powerful enough to allow guest code to perform valuable interactions, while at the same time preventing malicious or accidental damage or intrusion. The ADsafe subset can be verified mechanically by static tools like JSLint [14].

ADsafe was initially developed to host untrusted advertising content safely on a webpage. xBook's isolation mechanism is designed with the code base taken from an earlier version of ADsafe. We customized ADsafe by adding code for our component confinement model and mediation based on our labeling model, to prevent information

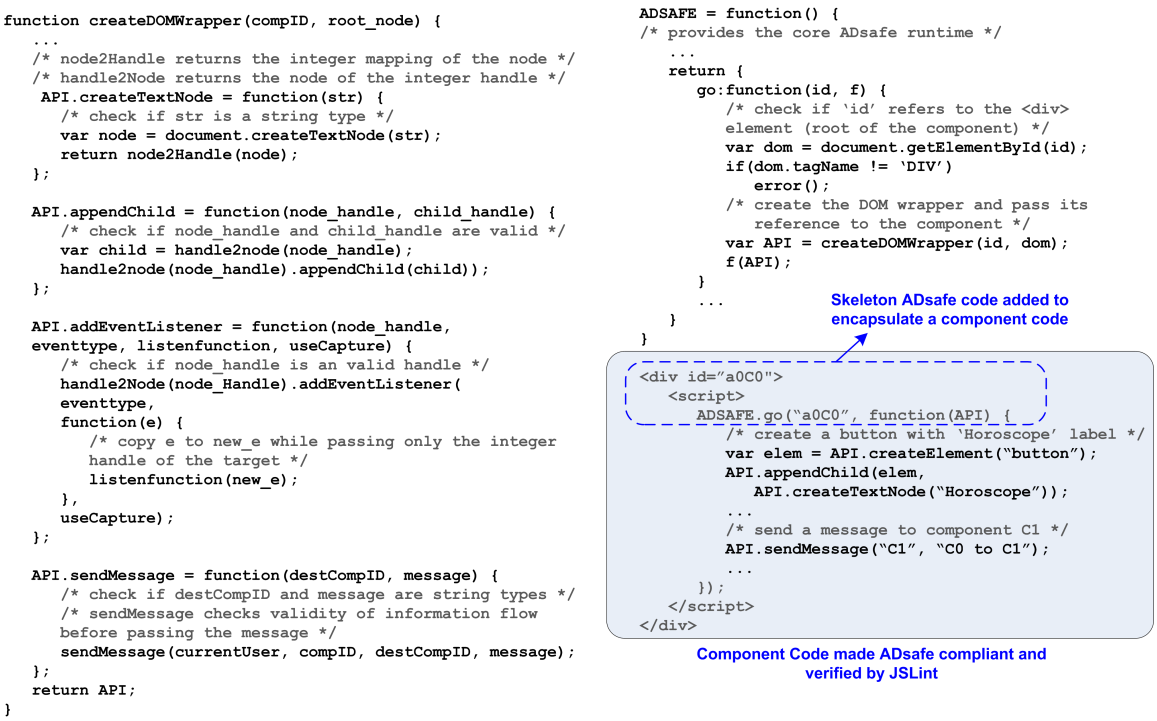


Figure 10: DOM wrapper implementation with sample functions.

leaks from the "sandboxed" application components. A recent version of ADsafe have since implemented some of our features, but still would require changes to be useful for our system.

One such example is that ADsafe runtime supports only a single level of confinement: all subtrees of the untrusted guest applications exist as children of the trusted web page code. One guest application does not have another guest application as a child to its subtree. In contrast, xBook design requires *nested* DOM subtrees that need to be isolated from each other. Figure 9 shows an example of a nested subtree, where component C_3 is a child of component C_1, which in turn is a child of C_0.

Our requirement is to restrict an application component to within a set of connected DOM elements that form the component. In the current DOM specification, any DOM element can parse through the tree of the page via its parent, children or siblings. We enforce confinement by providing the component elements only with a partial view of the page's DOM and only indirect access to the DOM objects.

Confinement Rule 1. One DOM element belonging to an application component should only access another DOM element of the page (that includes accessing its properties, adding a new element to it, etc.) iff they both belong to the same component.

As part of the implementation, xBook associates each component with a unique *DOM wrapper* object at the time of creation. Figure 10 shows the partial code of our DOM wrapper implementation. Before deploying an application, xBook verifies that each component code is ADsafe compliant. The code must be wrapped in a `<div>` element having an identifier, which forms the root of the component. xBook ensures that this identifier is unique to the application page. The `ADSAFE.go` method gives the component code access to the `API` object that maps to our DOM wrapper object. The `ADSAFE` code ensures that the second parameter passed to the `createDOMWrapper` function is equal to the identifier of the encapsulating `<div>` element, effectively preventing the developer from faking the identity of the components. It also ensures that the DOM wrapper instance gets the right identity of the component's root node.

The wrapper allows an untrusted component to view DOM nodes simply as integer handles; the component has no direct access to the real DOM. To read or modify the DOM, the component code passes the appropriate handles to the wrapper DOM object using the xBook APIs, which in turn interacts with the real DOM. Additionally, element creation and modification are administered using this component-specific wrapper object. For example, `createTextNode` method in Figure 10 would return an integer handle. Since a wrapper instance is identified by its root element `<div>` that is unique, the DOM wrapper object restricts the untrusted component code to interacting only with the portion of the document tree that belongs to that component. All direct accesses to any real DOM elements are forbidden: the wrapper is the only interface for accessing the elements and it is mediated by the xBook platform.

3.3.1.1 Event Handling

Another possibility of an application breaking the confinement mechanism originates from the way event handling is designed in the current DOM specification.

Every event has a target, i.e., the XML or HTML element most closely associated with the event. An event handler is a piece of executable code or markup that responds to a particular event. Any element of the DOM can register an event handler to receive a particular event type. Since an event generated from within a component can be received outside the component, the flow of events within a DOM needs to be controlled by the xBook platform for any potential leaks.

In the current DOM implementation, it is possible to assign multiple handlers for a given event. It allows a DOM element to capture events during either of the two phases in the event flow. The event flows down from the root of the document tree to the target element in the first phase called *capture*, then it bubbles back up to the root in the *bubbling* phase. An element can receive the event only if it lies in the path between the document root and the event target.

One of the goals of our event handling model is to keep the functionality of the current DOM model (including preserving the concept of the two stages). Therefore, we specify our event flow model as follows: for any application component, an element can receive an event iff it lies in the path between the *root of the component* and the target element for the event. We still need to restrict this access to a single component so that no outside component can receive the event; we provide such a restriction by the following confinement rule:

Confinement Rule 2. A DOM element belonging to an application component can receive an event iff the event target belongs to the same component.

We implemented our event handling model using the DOM wrapper object introduced in the previous section. As shown in Figure 10, the object makes a wrapper to the event handling interface available to applications. The wrapper receives the

event from the browser's DOM implementation and filters the information presented in the received event object before passing the event to the applications. Any information about the real DOM elements, such as the handler to the target element, is filtered; this prevents application's component code from breaking the confinement. The `addEventListener` method copies the received event `e` into `new_e` while transforming the real DOM element references to wrapped integer values. The xBook platform mediates the event delivery and as a result, ensures that an event can only be received by elements that belong to the same component that contains the target, thereby enforcing the second confinement rule.

3.3.2 Communication with External Entities

It is common for the applications to communicate with external parties to perform specific tasks. One typical example is the use of Google map APIs to generate maps of some address known to the application [15]. In other cases, a user's date of birth is used by applications to contact external providers to generate horoscopes [7]. What we achieve in our architecture as compared to the existing social networking platforms is that *we enforce the applications to make these communications explicit* so that more informed decisions can be made. The user or the platform can decide on the policies regarding which external entities are allowed to receive what piece of the user's private information. These policies could be coarse-grained for all applications of a user or fine-grained specific to each application. xBook ensures that the information flows from a specific application component to an external entity according to the defined policies.

There are two kinds of communication flows that can happen in our system:

Symmetric communication in which the response is received by the requesting component. This is a typical case for most client-server communication in which there is a two-way exchange of information between the two parties.

Asymmetric communication in which the response is not received by the component that made the request, but is handled by another component of the application. Our motivation for supporting this type of communication is to enable some specific application scenarios. One motivating example is the advertising scenario where advertisements are generated by external parties based on the information passed to them: Google generating advertisements based on the address passed to it. These external party advertisements are typically in the form of links that users click to access the related site. If we design this scenario using symmetric communication, these advertising links would not work, since the receiving component has been restricted to communicate only with Google and not any other party. In order to solve this problem, we can create another application component that is considered part of Google's trust domain; since Google servers are unconfined or public from xBook's point of view, the created component is also unconfined. We do not allow any other application component to peek into this new component or disrupt its integrity. Since we are only showing Google's view in this component and the application is not allowed to change this component, this component maintains the trust level of Google. The new component is placed in an `iframe` with its own DOM and hence cannot communicate with any other component. However, since the component is unconstrained, it is allowed to communicate with any external entity and as a result, the advertising links would work.

3.3.3 Communication between Components: Message Passing Interface

xBook exposes a one-way message passing API that the components use to pass messages to other components. We implement this interface using the DOM wrapper object as shown in Figure 10. The platform mediates this communication and ensures that the information flow model is enforced. Since each component is associated with a unique wrapper object that is used to send the message (Section 3.3.1), the sending

component of the message can not fake its identify to fraudulently pass the information flow checks: as seen in Figure 10, the value of `currentUser` and sender's `compID` are implicitly provided by the wrapper object to xBook's `sendMessage` function. A component can register a message listener with the platform through the xBook API. Any message intended for a particular component is delivered to its message listener. Since the platform knows the identity of each component, it makes sure that the message is delivered to the right component.

The purpose of our message passing interface is to allow xBook-mediated communication among untrusted components of an application, while still preventing creation of any hidden channels. To this end, we needed to evaluate some of the features of JavaScript that gives application writers alternatives to pass hidden information in the messages.

JavaScript is a weakly typed language and allows any property to be added to any object. For example, an object `message` can take a property `foo` using `message.foo = value;` where `value` could be a number, string or any other object type. Since all application components run in the same scope, a component can pass information to another component if it has access to an object of that component. Let us assume that a component C_1 is allowed to talk to another component C_2 as per the information flow policies, but C_2 can not communicate to C_1. Effectively, we have a one-way communication channel from C_1 to C_2. If C_1 passes the object `message` to C_2, the platform can observe `message`, but cannot identify the object handler `foo` being passed. C_2 can pass information to C_1 by writing to this handler.

We counter such leaks by limiting the message passing to being a JSON container [13], that is pure data. A JavaScript JSON container is a collection of key/value pairs or an array of values. These key/values are limited to pure data types such as string or numbers. We make a copy of the JSON object and pass the copy to guarantee that there are no additional properties in the passed object. This solution is

also effective against attacks by a message sender that use getters and setters.

The simplest way of designing the message passing interface is to pass messages from a source to a destination in a single thread of execution. This option opens up the possibility of a covert communication channel from a more restricted to a less restricted component. For example, let us consider that a less secret component C_0 is passing multiple messages to a more secret component C_1. Because of the single-threaded non-preemptive nature of JavaScript, C_1 will complete processing the first message before the control goes back to C_0. This creates a covert timing channel from C_1 to C_0. The amount of time taken by C_1 can be observed by C_0 and C_1 can change this time to pass the desired information bits to C_0.

We reduce the effect of this timing channel by making the message passing interface asynchronous. We achieve asynchronous behavior by implementing a global queue for message passing that is shared among all the components of an application. The receiving components register listeners with the platform in order to receive messages. A timer event dequeues an available message and delivers it to the message listener of the target component of the message. Note that addressing all covert channels in our system is beyond the scope of this thesis; we discuss this further in Section 4.4.

3.4 Server-side Components

The server-side of the application contains the main functionality for typical applications. It follows a familiar web server model where a server-side component is instantiated for every client request.

Besides the regular user-specific components on the server side, there are certain components that are user independent and works on non-user data or user public data. These components perform two tasks: First, they communicate with external parties to provide functionality independent of the user data. Second, they handle

statistical aggregation on user data sets. We discuss declassification based on data anonymization in Section 3.4.2.

The server components also protect application proprietary data that needs to be declassified before sending it to the client. The threat model is reversed in this case: the applications do not trust the user for their data, so they protect their internal data from being leaked to the users. For example, an application might be giving horoscope predictions to users based on their birth date, but it wants to protect the data or algorithm used for such predictions.

There is no direct communication between the server-side components: all such communication happens via application-specific storage. The platform ensures that the information flow is enforced while accessing the database. The platform also administers the communication with external parties and client-side as allowed by the labeling system.

3.4.1 Component Confinement

The server-side components need to be isolated from each other. The server-side of xBook mediates all communication flowing in and out from these components. There are several options available for server-side isolation. Operating system isolation mechanisms [25, 104] can be used to sandbox the application components. Another option is a language level confinement similar to the client-side isolation with options like Caja (JavaScript) [91], ADsafe (JavaScript) [2] and JoeE (Java) [59] available. We use ADsafe on the server-side in order to have the same language for developing application components for both client and server.

To the best of our knowledge, we are the first ones to port ADsafe to the server side. We had to make some modification to the ADsafe object to implement our server-side xBook APIs and to perform checking of the information flow labels. Each

server-side component holds a unique handle to the modified ADsafe object, and access is restricted to the set of APIs provided by the modified ADsafe object. The modified ADsafe object is conceptually similar to the DOM wrapper object on the client side, but is customized to work in the server-side environment. The platform verifies the validity of the information flow before any access is granted. The JavaScript execution environment is provided by Helma [11], a popular open source web application framework.

3.4.2 Anonymized Statistics

xBook ensures that no user data is leaked against the user's policies. A particular instance of an application can only have access to profile data that belongs to the user and only his friends. Different instances of the applications cannot share data due to the restrictions posed by xBook's labeling system.

It is desirable for some applications to have a view of all its users so that some statistical results can be published for the whole application. In other words, a component of the application needs to receive data of all the application users and still should be able to share these statistics as output to all users, crossing the boundary of friends.

In order to facilitate this case, we are exploring a three-step anonymization algorithm that provides conservative access to data for the applications. Currently, case 1 and 3 have been implemented, case 2 will be explored as part of our future work.

Case 1. If an application component requests a single field of user information for all application users, it is given access to the requested set in an unmodified form, but in a random order of sequence.

Case 2. If an application component requests multiple fields of user information for all application users, it is given access to the requested set in a form generated by anonymizing the original dataset and then randomizing the resulting tuples' order

of sequence. We plan to leverage some of the existing work [40, 89, 112] to generate the anonymized statistics. We acknowledge that providing security in anonymity and statistical queries is a challenging problem and has its own limitations [30, 89]. Addressing these limitations is orthogonal to our work and is not the focus of this thesis.

Case 3. Applications can also request the xBook platform for statistics on unanonymized data. This gives the applications more accurate statistics as compared to case 2, where some fields might be filtered or altered to preserve anonymity. xBook provides a limited list of such operations, including aggregation, maximum and minimum value over one or multiple fields.

Discussion. Anonymizing the data might limit some applications that rely on the original data for their functionality. One such example is an application that plots the location of a user's friends on Google maps, and would need to pass names and addresses of the user's friends to Google. The application also makes subsequent queries to Google (for example, to build a Google calendar of friends' birthdays). If the data is anonymized, the application might not produce completely accurate results.

On the other hand, if Google is provided with unanonymized data, it can use the data to cross-reference and identify the friends. This is a conflict between privacy and functionality. If functionality is preferred and unanonymized information is passed to external entities, user's personal information can be leaked. In such a case, our xBook design, at the minimum, enforces the applications to explicitly declare all external communication (including the data that will be transferred). Based on such information, the user can make a much more informed decision about adding the application.

3.5 Labeling Model

The xBook platform tracks and enforces information flow using a labeling system defined based on existing models [51, 82, 98, 140]. All system abstractions are layered on top of two types of entities – active and passive. Application components represent active entities that actively participate in label compatibility checks; database entries are passive entities. Every active entity corresponds to a principal and a label; passive entities only have a label.

We do not enforce information flow at the language level [98], but instead at the level of application components and database entries. There are multiple reasons for this choice: (1) it is simpler for the application programmers as they do not need to learn a new language or perform fine-grained code annotations, (2) information flow on a language like JavaScript with dynamically created source code may not be feasible, and (3) run-time information flow at fine-grained language level would probably be expensive as compared to a much coarser level of components.

The label specifies the secrecy level of an entity. It represents what information is contained in a passive entity and what information the active entity currently has or will read. The entity's principal defines whether the entity has declassification privileges over the label. xBook labels originated along the lines of the language based labels in Jif [98]. Labels represent the confidentiality or secrecy level of an entity in the system. Integrity labeling is not the focus of this work since we are focusing on privacy.

A label L is represented as a set of tags, with each tag having one principal as owner o and another set of principals called readers $R(L, o)$. The owner is the principal whose data was observed in order to construct the data value. The readers represent principals to whom the owner is willing to release the information. An example of a typical label is $L = \{o_1 : r_1, r_2;\ o_2 : r_2, r_3\}$, where $O(L) = \{o_1, o_2\}$ denote the owner set for the label and readers sets are $R(L, o_1) = \{r_1, r_2\}$ and $R(L, o_2) = \{r_2, r_3\}$.

In the xBook system, principals represent the identities of various entities in the labeling model. There are five types of principals in our system:

- $C(a_i, u_j)$ and $S(a_i, u_j)$ represents the client-side and server-side components for an application a_i specific to a user u_j.
- $C(a_i)$ and $S(a_i)$ represents user-independent client-side and server-side components for an application a_i.
- u_j represents the entities that the user u_j is in complete control of. Once the user u_j is logged into the xBook system, the user's browser is assigned the principal u_j.
- $\top$, $\bot$ where $\top$ is highest priority principal in the system and is allotted to the xBook platform. For the sake of completeness, $\bot$ is the least privileged principal.
- External entities also have principal names that contain the hostname and optionally the scheme and port (like in URLs). For example, `https://www.example.com:8888` represents one such principal.

Our model assumes static labels for the entities and information flows from one entity to another if allowed by the label comparison of the end points. Information can flow from one label L_1 to another label L_2 only if L_2 is more *restricted* than L_1 denoted as $L_1 \preceq L_2$.

Restriction. $L_1 \preceq L_2 \iff O(L_1) \subseteq O(L_2)$ and $\forall o \in O(L_1), R(L_1, o) \supseteq R(L_2, o)$

3.5.1 acts-for Hierarchy

To facilitate easier conversion of user policies to low-level labels, system entities are statically labeled. We decided on immutable labels since it improves usability of the application programming model from the perspective of the application programmer. Unexpected runtime failures can occur when labels of components change at

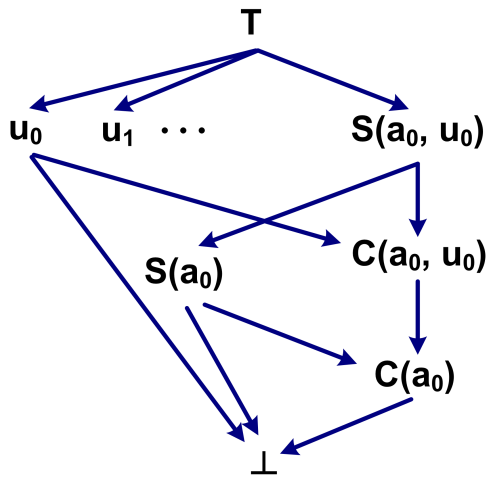


Figure 11: Label hierarchy in `xbook`.

runtime [82]. With immutable labels one can statically verify that all the communication dependencies with respect to other components, external entities, storage will be satisfied.

Some principals have the right to act for other principals and assume their power. The acts-for relation is transitive, defining a hierarchy or partial order of principals [51]. The right of one principal to act for another is predefined by the platform. Figure 11 presents the acts-for relationship within the xBook system. This hierarchy defines the priority of different principles in the system. The reasoning behind the defined hierarchy is as follows:

- $\top$ defines the xbook platform and has the highest security label. As a result, it can declassify any label.

- Any data sink or source that is not explicitly defined by xBook is modeled as an unprivileged entity with label $\bot$.

- The client-side components are given lower priority than server-side components, because intuitively server-side components residing on xBook servers are more trustworthy than client-side components. For example, $S(a_0, u_0)$ has higher priority over $C(a_0, u_0)$ for application a_0 and user u_0. The server-side

Algorithm 2 Label Compatibility Check Algorithm.

$eL_1 = (entity_1 \text{ is a database}) \ ? \ L_1 : \text{maxDeclassify}(L_1, P_1)$
$eL_2 = (entity_2 \text{ is a database}) \ ? \ L_2 : \text{maxRestrict}(L_2, P_2)$
if $eL_1 \preceq eL_2$ **then**
  ALLOW flow from $entity_1$ to $entity_2$
else
  DENY flow
end if

Figure 12: Algorithm to check if the information flow from $entity_1$ to $entity_2$ is allowed.

components can declassify an application's proprietary data, which has been labeled in a manner such that it cannot be directly read by client-side components.

- User-independent principals are at a lower priority than any user-specific principal. This allows user-specific components to read user-independent data generated by an application, also effectively allowing users to read statistical data generated for the whole application.

- Principals representing the end user are higher than the corresponding client-side principals since the user controls the client.

3.5.2 Flow Enforcement

Information flows within the xBook system if the label of source is less restricted than that of destination. Such flow restrictions have been proposed earlier in classical information flow control models [34]. We introduce the concept of endpoints similar to the Flume model [82]. Instead of changing the labels of the entities, for every communication the source and the destination create an endpoint each to facilitate the flow. The entity, based on its principal, can restrict or declassify its label and allocate it to an endpoint for communication. While restricting a label means adding

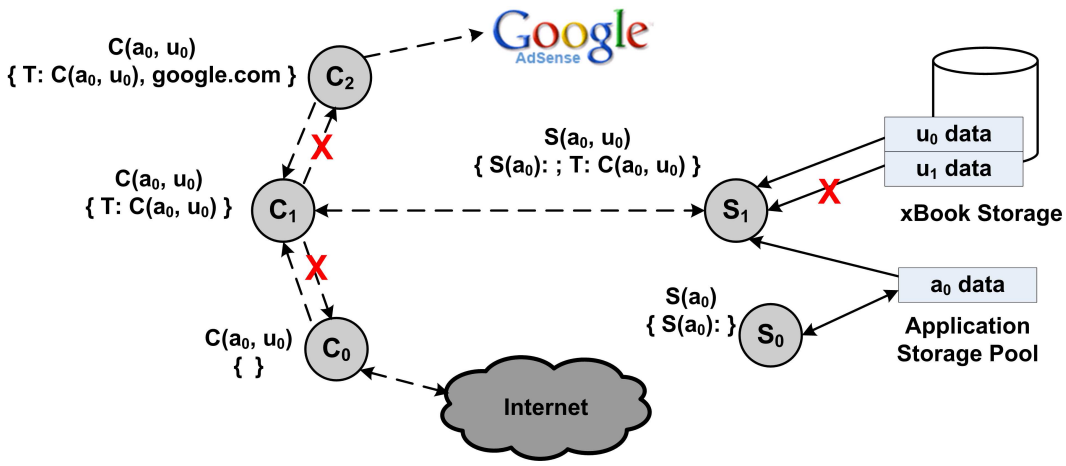


Figure 13: Typical Flows in `xBook` system with the corresponding labels. For every component, the first parameter is the principal and the second is the label associated with the component.

more owners and removing readers, declassification either adds some readers for an owner o or removes the owner o. This relabeling can be done only if the principal of the entity is higher than an owner o in the hierarchy.

Figure 12 shows our flow enforcement algorithm, where maxRestrict and maxDeclassify are defined as:

- ***maxRestrict(L, P).*** $O(L) = O(L) \cup descendent(P)$; $\forall o \in descendent(P)$: $R(L, o) = \{\}$
- ***maxDeclassify(L, P).*** $\forall o \in O(L)$: if $(o \in descendent(P))$ then $O(L) = O(L) - \{o\}$

where $descendent(P)$ represents all descendents of a principal P in the acts-for hierarchy, $O(L)$ is the set of owners for label L and $R(L, o)$ represents a set of readers in label L for owner o. Intuitively, the communicating end points support the communication with the sender declassifying its label to the maximum possible using $maxDeclassify$ and the receiver restricting its label using $maxRestrict$. Since the information can only flow from a less restricted to a more restricted component, these functions facilitate the flow of information.

Some typical flows in the xBook system are depicted in Figure 13. To demonstrate the validity of our algorithm, let us consider the example of the flow between the client-side component C_1 and the server-side component S_1. For the flow from S_1 to C_1,

$$\begin{aligned} eL_1 = \; & maxDeclassify(\{S(a_0) :: \top : C(a_0, u_0)\}, \\ & S(a_0, u_0)) = \{\top : C(a_0, u_0)\} \\ eL_2 = \; & maxRestrict(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ = \; & \{C(a_0, u_0) :: C(a_0) :: \top : C(a_0, u_0)\} \end{aligned}$$

Recollecting the definition of restriction, we can see that $eL_1 \preceq eL_2$, therefore S_1 can send data to C_1. Considering the reverse flow from C_1 to S_1,

$$\begin{aligned} eL_1 = \; & maxDeclassify(\{\top : C(a_0, u_0)\}, C(a_0, u_0)) \\ = \; & \{\top : C(a_0, u_0)\} \\ eL_2 = \; & maxRestrict(\{S(a_0) :: \top : C(a_0, u_0)\}, S(a_0, u_0)) \\ = \; & \{S(a_0, u_0) :: S(a_0) :: C(a_0, u_0) :: (a_0) :: \\ & \top : C(a_0, u_0)\} \end{aligned}$$

We can see that $eL_1 \preceq eL_2$, i.e., C_1 can send data to S_1. Effectively, there is a two-way communication between C_1 and S_1.

3.5.3 Case Study: Horoscope Application Lifecycle

An application's lifecycle consists of three steps: the application being hosted by xBook, a user adding the application and then the user accessing it.

Hosting. Before xBook accepts a new application, the developer needs to provide the following information:

- The application provides the components to be deployed, in each case specifying if the component is client-side or server-side and if it is user-dependent or not,

what user data would the component require and which external entities and other components will it communicate with. In our horoscope example, there are three components: S_0 communicates with `www.tarot.com` and requires no user data; S_1 requires user's birthday; C_1 is on the client-side and also requires user's birthday.

- The application also states that there are user-independent or user-dependent storage pools and each is named declaratively by the application. This ensures that the storage pool names do not leak any user information, as the application has no user information at this time. For example, horoscope application declares a storage pool for storing its application data generated by S_0.

Based on the label of the user data, xBook derives the labels and the principals of the components. The birthday field has a label $\{\top : C(a_i, u_j)\}$, therefore the following labels are allocated to the horoscope components:

- S_0 *Principal:* $S(a_i)$, *Label:* $\{S(a_i) : \}$
- S_1 *Principal:* $S(a_i, u_j)$, *Label:* $\{S(a_i) : ;$
  $\top : C(a_i, u_j)\}$
- C_1 *Principal:* $C(a_i, u_j)$, *Label:* $\{\top : C(a_i, u_j)\}$

The principals define if the component is server-side or client-side, and if it is user-dependent or not. The labels allow S_1 and C_1 to read the birthday field. S_0's label allows it to declassify itself to be public to communicate with `www.tarot.com`, and write to the storage pool that is given S_0's label. The storage pool label prevents any of the client-side components (C_1) from viewing this data, thereby protecting application data from untrusted users. S_1 is allowed to read from the storage pool. The labels of S_1 and C_1 correspond to the labels of S_1 and C_1 respectively in Figure 13, where $i = 0$ and $j = 0$. As we have observed in the last section, the labels of S_1 and C_1

effectively allow a two-way communication channel. Thus, S_1 can pass the results to C_1 that, in turn, can present a formatted form of the horoscope to the user's browser.

Application Addition. When the user is adding the application, he is provided with a manifest that declares what information is passed to which external entity. xBook derives the manifest from the component information provided by the application developer. For example, since none of the components of the horoscope application share any user information with any external entity, horoscope's manifest would specify that it does not pass any information to any external entity. Since the user's birthday is not shared with any external entity, the application does not need to declare its need to access the birthday information.

Application Access. When the user is accessing an application, all user-specific components are instantiated for that user, replacing the user wildcard in the template of labels and principals with the user identifier. This enforces access control across multiple users: access is only granted if it is aligned with the user's privacy policy, for example, access is granted to only user's friends.

3.6 Evaluation

3.6.1 Prototype System and Example Applications

We developed a working prototype of the xBook system, which includes platform code and APIs for developing third-party applications. We also implemented the labeling model that enforces information flow control for the data flowing through the system and prevents any information leaks. Our xBook platform consists of about 4300 lines of JavaScript code.

We developed two sample applications using the xBook APIs (given in Table 5) to show the ease and viability of application development in xBook. These applications are similar in functionality to two popular Facebook applications: Horoscope [7] and TopFriends [22].The horoscope application produces a user's daily horoscope based

on his birthday information. The utility application based on TopFriends produces a customized profile for the user based on his complete profile information. It also generates a Google map showing the user's home location on the map. The applications are written in JavaScript using xBook APIs, with the horoscope application having about 180 lines and the application based on TopFriends having around 480 lines of code. We tested these applications against a series of synthetic scenarios, where these applications tried to leak the user's private information. Our tests showed that the xBook system was successful in detecting and preventing all such leaks.

3.6.2 Porting xBook on Facebook

In order to show the practical viability of the system and to demonstrate that our system can be incrementally deployed, we ported the xBook platform as an application on Facebook. Since Facebook allows any application to have access to user data, including their friends' data, of any user adding the application, xBook as an "application" is able to receive the data of the users agreeing to use the xBook platform. Applications developed using xBook APIs can execute on top of xBook, while still running on xBook servers. Since xBook act as an application for Facebook, xBook's response would be rendered as part of Facebook's web page. Since the third party applications are encapsulated in the page forming xBook's response, the output of these applications would also be effectively rendered on Facebook (Figure 6(c)). Facebook provides the data feed to xBook, which then enables access to this data for xBook applications in a controlled manner through xBook APIs. Facebook's user identity is maintained within xBook.

We envision xBook to be assimilated into the Facebook platform with Facebook providing two levels of application service. First, the current applications based on current Facebook design would be supported. Second, applications that are developed using xBook APIs are supported, with added privacy protection advantage. Users

can be given the discretion to choose between the two options, and the users' choice can drive new application development on xBook.

3.6.3 Security Analysis

Our analysis shows that xBook prevents the applications from leaking any user information. All of the documented leaks in the current social networks are prevented in the xBook system. For example, the TopFriends leak [92] cannot happen in our system because a separate application instance is created for every user. Each instance only has view of the data accessible to that user and xBook mediates all cross user data accesses.

We evaluated the privacy protection ability of our system in three steps. First, we analyzed the security of the xBook design in view of the potential leaks specified in the formal model (Section 3.2.2). Second, we developed a set of synthetic attacks targeting the xBook framework and performed experiments to show that our prototype successfully prevents these attacks. Finally, we prove that xBook's information flow model ensures that information leaks cannot happen in the xBook design.

We first analyze the security of our prototype and show that all the attacks discussed in Section 3.2.2 will not succeed against our design. Attack type A1 is prevented due to the various mechanisms developed in our system for client-side confinement (Section 3.3.1), such as component isolation, event handling, etc. A2 is prevented by server-side confinement of application components, only allowing them to communicate via storage. Leaks via A3 and A4 are inherently prevented by mediating the information flow from the database to application components with label enforcement based on user-defined policies, and also by anonymizing data for statistical purposes (Section 3.4.2). A5 is also prevented by label enforcement before the client-side request is passed to the server-side component and before response is returned. Enforcing the confinement model to mediate the external communication,

Table 3: Prevention of information leaks against various kinds of synthetic attacks.

Attack Step	**Attack Type**	**Prevented by xBook?**
One client component accessing another component's DOM object	A1	√
Leaks via the message passing interface	A1	√
A component creates or destroys a less restricted component leaking information	A1	√
Retrieve information of another user not in the friend list	A3/A4	√
Client component retrieves more restricted information from the server	A5	√
Leaks to an unknown external entity	A6/A7	√
Leaking restricted information to an allowed external entity	A6/A7	√

both in synchronous and asynchronous communication scenarios, prevents A6 leaks (Section 3.3.2). Following the same lines, A7 is prevented on the server-side.

Second, we tested the ability of our prototype by creating synthetic exploits that try to break out of xBook's information flow control model to leak user information. We developed a sample application to launch these attacks against our prototype; if successful, these attacks allow the application to leak information to entities outside the system. Table 3 contains the results of testing our prototype against a wide range of these synthetic attacks. In all our experimental tests, xBook successfully prevented the leaks before the information could be passed outside the system.

We can also prove that if xBook's confinement mechanism is correctly enforced, the information model ensures that no user information is leaked to external entities (Theorem 1) and to any other user (Theorem 2) outside the user-defined policies.

Theorem 1. *Given a set of policies $P = D \times X$, where the application can pass user's information field $d \in D$ to external entity $x \in X$, and assuming that the intended confinement is enforced, the information flow model ensures that there is no possible*

leak outside the xBook system. In other words, if $(d, x) \notin P$ *then* $\forall C_i : C_i \nrightarrow^d x$*, where* C_i *are application components and* $C_i \nrightarrow^d x$ *shows that* C_i *can not pass data item* d *to* x*.*

Proof. Let $C^0, C^1, \cdots C^k$ represents the information flow path of a data element d from the xBook database to external entity x.

We present the proof by contradiction. Let us assume that C^i can pass any information (represented by $*$) to x, illustrated as $C^i \xrightarrow{*} x$. This communication is monitored by our xBook platform, but the platform does not know the semantics of the information being passed.

Also, $\forall i \in [0, k] : C^{i-1} \xrightarrow{*} C^i \implies L^{i-1} \preceq L^i$ (flow is a restriction)

$C^i \xrightarrow{*} x \implies L^i \preceq L^x$

Therefore, $L^{i-1} \preceq L^x \implies C^{i-1} \xrightarrow{*} x$

Continuing this by induction, $C^0 \xrightarrow{*} x$

In our labeling model, the computational granularity is at the component level. Therefore, we consider that $\forall C_i : Output(C_i) = F(Input(C_i))$ for any computation F.

For component C^0, $Input(C^0) = d$, $Output(C^0) = * \implies * = F(d)$

Since the input to C^0 is supplied by the xBook platform, and since $(d, x) \notin \mathbb{P}$, $C^0 \nrightarrow^* x$.

This is a contradiction. Therefore, $C^i \nrightarrow^* x$.

By definition, $*$ represents any information (including d). Therefore, $C^i \nrightarrow^d x$.

Theorem 2. *Given a set of user policies* $P(x) = D \times U$*, where the application can pass user* $x \in U$*'s information field* $d \in D$ *to another user* $y \in U$*, and assuming that the intended confinement is enforced, the information flow model ensures that user-user access control is enforced in the xBook system. In other words, if* $(d, y) \notin P(x)$

Table 4: Performance results of various operations in typical xBook applications.

Application	User latency	Server processing time	Time for label checks (Number of checks)	Over-head
Horoscope	183.1ms	128.8ms	7.7ms (6)	4.2%
Map utility	111.4ms	51.2ms	3.5ms (2)	3.1%

then $\forall C_i(x), C_j(y) : C_i(x) \nrightarrow^d C_j(y)$*, where* $C_i(x)$ *and* $C_j(y)$ *are components of application instance for user* x *and* y*, respectively.*

Proof. Similar to Theorem 1.

3.6.4 Performance Estimates

xBook does not impose a substantial burden on the performance of the third party applications. With an architectural framework of developing applications, it is difficult to accurately predict the impact of our design on the performance of these applications as perceived by the user. To get a rough estimate of the cost of supporting the xBook design and the overhead involved in our system, we conducted some experiments with our sample applications, measuring latency at the user end and overhead imposed by the mediating design of xBook.

The xBook server side is hosted on a 2.4GHz Pentium 4 machine with 512MB of RAM. The requests are made from Firefox 3.0 browser on a 2.33GHz, 2GB RAM, Pentium Core Duo laptop. Each test was run 10 times and values were averaged. We define user latency as the difference in the time when the request is made at the browser and the time at which the response is received by the browser. Table 4 shows the time required by xBook's information flow control in comparison to the user's overall latency. Server processing includes the application's logic, database access to retrieve required user data, and xBook flow checks, and is independent of the network latency experienced by the application. We instrumented our code to derive the time for performing label checks in the system, and measured overhead as a function of the

label checking time over the total latency experienced by the user. Our results show that the overhead introduced by xBook's label checks is considerably small: about 4% for the horoscope application and 3% for the map utility marking user's hometown location on Google maps.

On a cluster of commercial servers with much better computational capacity, these values will be even smaller. Although it is not possible to precisely determine the cost of our approach without a large scale experiment, both the details of our design and the results from these experiments, support the conclusion that xBook design would not substantially increase the latency experienced by users.

3.7 Discussion

In this section, we discuss the limitations of the application design in xBook and address some of the challenges arising from the new requirements imposed by our design.

Our xBook design imposes no limitations on applications that follow a "pull model", i.e., xBook would preserve the functionality of applications that only receive data from external entities without passing any private information to these entities. Our horoscope application is an example of such as application: one public component of horoscope pulls horoscope data from `www.tarot.com` and does not pass any of the user's profile information. Note that the xBook platform does not need to sanitize the request parameters (in both GET and POST requests), as the component making such requests has no user information that can be leaked. Another component, which has access to the user's birthday information, uses the data to calculate the daily horoscope corresponding to the particular user. This component has no communication with any external entity.

On the other hand, our design might limit some of the applications that require data to be sent to external entities for receiving user-specific information. One typical

example is the use of Google APIs to generate maps: it requires a location to be passed to Google before the map is generated. In many cases, we expect these external entities to be larger and well branded entities, such as Google, Yahoo, etc. Such cases could be whitelisted after explicit approval from the user. Note that xBook makes no recommendation about which websites can be trusted, including Google and Yahoo; such trust decisions are made by an individual user from his own knowledge and experiences. Our xBook system can keep track of these approvals across applications for every user, so the users need to approve an interaction only once.

Any social networking application would follow either the pull model or the push model to get data from external entities. In both cases, our platform enforces the applications to make all such interactions explicit and allows the user to make a more informed decision based on the information available. We argue that an application using the pull model would be more acceptable to the users as it requires minimal trust decisions from a user's perspective. It is possible to transform many of the current social networking applications that use the push model to start using the pull model. We acknowledge that such a transformation would require some changes to the application design, and in some cases, such transformations might not be practical due to large download size of the required data. However, if enough users decide not to use the application in view of privacy concerns, it would motivate the developers to consider such a transition.

Our system also suffers from classical covert channels, e.g. timing, memory, process, etc. However, in general these channels have limited bandwidth and viable approaches such as randomizing the time (for example, the delivery time of our message queue discussed in Section 3.3.3) can further limit their utilities. We plan to study some of these channels as part of our future work.

Scalability of the applications is not a concern in our system: applications hosted on clusters outside xBook would now be hosted on clusters inside the xBook platform.

The application developers are already paying for hosting their applications, in most cases to third-parties or cloud owners like Amazon EC2 [3]. Thus, instead of the developers paying to these parties, they would be paying to xBook for the hosting service. xBook, in turn, can outsource the hosting to third-parties, still assuming control of the hosted applications.

We also propose a hybrid model where only the application components that require access to xBook's private data needs to be hosted at the xBook servers. Other public components can be controlled by the application developers on their own servers. Such an approach is useful for many applications as research has shown that a large number of applications do not use any private data to perform their functionality [55].

3.8 Summary

We presented a novel architecture for a social networking framework, called xBook, that substantially improves privacy control in the presence of untrusted third-party application. Our design allows the applications to have access to user data to preserve their functionality, but at the same time preventing them from leaking users' private information.

We developed a working prototype of the system as an application on Facebook. We showed the viability of our system by developing sample applications using the xBook APIs: these applications are similar in functionality to the applications on existing social networks.

Our system shows promise in designing potentially valuable future applications, that would require user data to provide more customized service to the user. The growing popularity of social networks would attract increasing attention from attackers because of the value of user information available in these networks. This user information not only has commercial value, but when combined with some anonymized

public data such as medical records, might leak more sensitive information [99, 126]. The current design of social networking applications poses a serious threat to the privacy of individuals that needs to be mitigated; the xBook platform is a major step in protecting user privacy in social networking applications.

The concepts and designs provided in this chapter for social networking platforms would similarly apply to other application platforms. Any effective solution to enforce user-defined security and privacy needs to cover both the trusted applications and untrusted third-party applications. Our two frameworks described in the last two chapters, xAccess and xBook, together provide a comprehensive solution for user-specified security policies and enforcement.

Table 5: Set of `xBook` APIs exposed for application development.

`createComponent(compId, size, nodeHandle)`	create a new component as a child of the specified node
`destroyComponent(compId, nodeHandle)`	destroy the specified component
`sendMessage(destCompId, message)`	send message to the destination component
`addMessageListener(listenerFunction)`	add a message listener to the component
`contactExternal(entityURL, nodeHandle)`	invoke the given URL with response received as a child to the given node
`readFromAppDB(attrName)`	get the value of the given attribute specific to the application
`writeToAppDB(attrName, attrValue)`	set the value of the given attribute specific to the application
`getInfoFromServer(serverCompId, function)`	call the given function in the specified server-side component
`getStatsFromDB(fieldName, statisticsType)`	returns the results of statisticsType operation on fieldName
`getAnonymizedStats(fieldNameList)`	returns the anonymized values of fieldNameList
`createTextNode(text)`	create a text node
`appendChild(nodeHandle, childHandle)`	add a child node to the specified node
`removeChild(nodeHandle, childHandle)`	remove the child node of the specified node
`getFirstChild(nodeHandle)`	returns handle of the first child of the specified node
`getLastChild(nodeHandle)`	returns handle of the last child of the specified node
`getNextSibling(nodeHandle)`	returns handle of the next sibling of the specified node
`getPreviousSibling(nodeHandle)`	returns handle of the previous sibling of the specified node
`getParentNode(nodeHandle)`	returns handle of the parent node of the specified node
`getElementById(elementId)`	return the virtual handle to the element node only if it is within the component
`createElement(elementTag)`	create a DOM element of the specified type
`setAttribute(nodeHandle, attrName, attrValue)`	set the attribute to the specified value for the given node
`addEventListener(nodeHandle, eventType, listenerFunction, useCapture)`	add an event listener to the given node with specified arguments
`hasChildNodes(nodeHandle)`	specifies if the given node has any child node within the component
`getNodeType(nodeHandle)`	returns the type of the specified node
`getNodeName(nodeHandle)`	returns the name of the specified node
`getNodeValue(nodeHandle)`	returns the value of the specified node

CHAPTER IV

BROWSER POLICIES

The security of client-side software, specifically the web browser, is critical to achieve effective end-to-end security of web content. However, new security challenges emerge for the web browsers as a result of the new developments behind Web 2.0. Web browsers have gradually evolved from an application that views static web pages to a rich application platform on which mutually distrusting web site principals co-exist and interact [75, 128, 129]. Along the way, the browsers' access control policies have also been evolving, but unfortunately this happened in a piecemeal and ad-hoc fashion alongside the introduction of new browser features (such as AJAX) or resources (such as local storage). There have been no principles or invariants that a new access control policy must follow or maintain. Consequently, numerous incoherencies in browsers' access control policies exist, presenting hurdles for web programmers to build robust web applications.

In this chapter, we examine the current state of browser access control policies, uncover and analyze the incoherencies in these policies, and measure the cost of eliminating them in today's web.

An access control policy configures how a principal accesses certain resources. This involves defining how principals are identified, how resources are labeled with principal IDs, and how these labels may be changed and handled at runtime. Unfortunately, browsers often mismanage principals, resulting in access control inconsistencies. We focus on three major sources of these problems: inconsistent principal labeling, inappropriate handling of principal label changes, and disregard of the user principal.

Inconsistent principal labeling. Today's browsers do not have the same principal definition for all browser resources (which include the Document Object Model (DOM), network, cookies, other persistent state, and display). For example, for the DOM (memory) resource, a principal is labeled by the origin defined in the same origin policy (SOP) in the form of <protocol, domain, port> [111]; but for the cookie resource, a principal is labeled by <domain, path>. Different principal definitions for two resources are benign as long as the two resources do not interplay with each other. However, when they do, incoherencies arise. For example, when cookies became accessible through DOM's "document" object, DOM's access control policy, namely the SOP, undermines some of cookie's access control policies (Section 4.1.3.1 gives a more detailed analysis).

Inappropriate handling of principal label changes. A web application is allowed to change its principal's label at runtime through the use of the `document.domain` DOM property. Nevertheless, the access control state is often kept static and such "effective" principal IDs set by `document.domain` are disregarded. This leads to access control incoherencies.

Disregard of the user principal. In this work, we introduce the concept of a *user principal* in the browser setting. The user principal represents the user of a browser. Sometimes, the user principal is disregarded in existing browsers' access control policies. Certain resources should belong to the user principal *exclusively*. They include the user-private state such as clipboard and geolocation, user actions like navigating back and forward in browsing history, and a browser's UI including the current tab. These resources should not be accessible by web applications without user permission; otherwise, a web site could impersonate the user and violate user privacy. Unfortunately, today's DOM APIs expose some of these resources to web applications.

To systematically analyze and uncover the incoherencies created by these three

problem areas, we have devised a set of coherency principles and constructed tests to check major browsers (including Internet Explorer, Firefox, and Google Chrome) for violations of these principles and to uncover the incoherencies that ensue.

A major goal of our work is to evaluate the compatibility cost of removing unsafe browser features that contribute to the incoherencies. To this end, we have built *WebAnalyzer*, a scalable, crawler-based browser-feature measurement framework that can inspect a large number of web pages by rendering them in instrumented browsers. WebAnalyzer captures the DOM interactions of a page by interposing between the JavaScript engine and the DOM renderer, captures the protocol-level behavior through an HTTP proxy, and analyzes the visual appearance of a page by extracting its page layout.

Armed with WebAnalyzer, we have conducted measurements on the prevalence of unsafe browser features over the most popular 100,000 web sites as ranked by Alexa [27]. Our results pinpoint some unsafe features that have little backward compatibility cost and are thus possible to remove from current browsers without breaking many sites. For example, we find that most APIs controlling user-owned resources, descendant navigation, and incoherencies in XMLHttpRequest's principal labeling have low compatibility costs, whereas a substantial number of sites depend on "dangerous" functionality provided by `document.domain` or transparent cross-origin overlapping frames. Overall, we believe that by estimating the prevalence of unsafe features on the web, our measurements can guide future browsers to make better security vs. functionality trade-offs.

In summary, this work makes the following contributions:

- A systematic, principal-driven analysis of access control incoherencies in today's browsers.

- Introduction of the user principal concept for the browser setting.

- A comprehensive, extensible compatibility measurement framework.
- The first large-scale measurements on the compatibility cost of coherent access control policies.

Chapter Organization. The rest of the chapter is organized as follows. Section 4.1 presents our systematic analysis of today's browser access control policies and enumerates access control incoherencies. Section 4.2 discusses our measurement motivation, tools, and infrastructure. Section 4.3 presents our measurement results and gives recommendations on which unsafe policies can be eliminated with acceptable compatibility cost. Section 4.4 discusses limitations of our approach, and Section 4.5 concludes.

4.1 An analysis of browser access control incoherencies

In this section, we present our systematic analysis of today's browser access control policies and enumerate their incoherencies.

4.1.1 Methodology

For a systematic analysis, we establish the following access control coherency principles to guide our search for incoherencies:

1. Each shared browser resource, i.e. a resource shared among multiple principals, should have a principal definition (labeling of principals that share the resource) and have an access control policy.
2. For each non-shared browser resource that is explicitly owned by a single principal, the resource should have an owner principal with a specific label or be globally accessible.
3. When two resources interplay, both resources should have the same principal definition.

This is because when two resources have different ways of labeling principals and when they interplay, their respective access control policies can be in conflict.

4. All access control policies must consider the runtime label of the principals, namely, the "effective" principal ID.

5. The user principal's resources should not be accessible by web applications.

   This is because when the user principal's resources are accessible by web applications, the user's privacy may be compromised or a web application could act on the user's behalf without the user's knowledge.

We look for violations of these principles and check for incoherencies when violations take place. The pseudocode below illustrates our manual analysis process.

```
0 foreach (browser resources) {
1   if exists (access control) {
2     if !considers (effective principal ID)
3       check improper principal ID changes
4   } else
5     check if lack of policy is appropriate
6 }
7
8 foreach (pairs of resources) {
9   if (they interplay &&
10       the principal/owner labeling differs)
11     check resource interplay incoherencies
12}
```

For each resource, we check whether it has an access control policy. If not, we check whether the lack of policy is appropriate (line 5, for example, Section 4.1.5 illustrates on how some resources that belong to the user principal lack access control considerations). If yes, we further check whether the access control policy considers the effective principal ID that sites can change dynamically at render-time. If it does not, then we check for incoherencies there (line 3, Section 4.1.4).

In addition, we go through all pairs of resources; if they interplay and if they have the different principal definitions, we check for incoherencies (line 11, Section 4.1.3). Careful readers may wonder what happens to the interplay of more than two resources. Coherency in this context is a transitive property. That is, if a Resource 1 and Resource 2's access control policies are coherent (namely have the same principal definitions) and that of Resource 2 and Resource 3 are coherent, then the access control policies of Resource 1 and Resource 3 are also coherent since their principal definitions should also be the same.

The enumeration of resources is done by manually browsing through IE's source code (more in Section 4.1.2). Our incoherency checks are done through test programs on major browser versions.

Despite our effort to be comprehensive, it is possible that we miss some browser resources or miss some interplays among the resources. We hope our work to be a start for a community effort on mapping out the full set of browser access control policies.

4.1.2 Browser resources

In this section, we enumerate all types of browser resources. A browser resource may be shared among (some definition of) principals or may not be shared and is explicitly owned by some principal. Table 6 shows the shared resources and their respective principal definitions. Table 7 shows non-shared resources and their respective owners.

Table 6: Shared browser resources and their respective principal definitions. *Display access control is not well-defined in today's browsers.

Shared resources	**Principal definition**
DOM objects	SOP origin
cookie	domain/path
localStorage	SOP origin
sessionStorage	SOP origin
display	SOP origin and dual ownership *

Table 7: Non-shared browser resources and their respective owner principal. *Access control is not well-defined in today's browsers.

Non-shared resources	**Owner**
XMLHttpRequest	SOP origin
postMessage	SOP origin
clipboard	user*
browser history	user*
geolocation	user

We now describe each resource, their principal or owner definition, and its access control policy in turn.

A *DOM object* is a memory resource shared among principals labeled with SOP origins, namely, <protocol, domain, port>. The access control policy of DOM objects is governed by SOP [111], which mandates that two documents from different origins cannot access each other's HTML documents using the Document Object Model (DOM), which is the platform- and language-neutral interface that allows scripts to dynamically access and update the content, structure and style of a document [52].

A *cookie* is a persistent state resource. The browser ensures that a site can only set its own cookie and that a cookie is attached only to HTTP requests to that site. By default, the principal is labeled with the host name and path, but without the protocol and the port number [60, 81], unlike SOP origins. For example, if the page `a.com/dir/1.html` creates a cookie, then that cookie is accessible to

Table 8: Access control policy for a window's landlord and tenant (being a different principal from the landlord) on Gazelle, IE 8, Firefox 3.5, and Chrome. RW*: The URL is readable only if the landlord sets it. If the tenant navigates to another page, landlord will not see the new URL. W*: the landlord can write pixels when the tenant is transparently overlaid on the landlord.

	Landlord			**Tenant**		
	Gazelle	IE	FF/ Chrome	Gazelle	IE	FF/ Chrome
position (x,y,z)	RW	RW	RW		RW	
dimensions (height, width)	RW	RW	RW	R	RW	R
pixels		W*	W*	RW	RW	RW
URL location	W	W	RW*	RW	RW	RW

`a.com/dir/2.html` and other pages from that `dir/` directory and its subdirectories, but is not accessible to `a.com/`. Furthermore, `https://a.com/` and `http://a.com/` share the cookie store unless a cookie is marked with a "secure" flag. Non-HTTPS sites can still set "secure" cookies in some implementations, but cannot read them back [31, 76, 138]. A web programmer can make cookie access less restrictive by setting a cookie's `domain` attribute to a postfix domain or the path name to be a prefix path.

Local storage is the persistent client-side storage shared among principals defined by SOP origins [72].

Session storage is storage for a tab [72]. Each tab has a unique set of session storage areas, one for each SOP origin. The *sessionStorage* values are not shared between tabs. The lifetime of this storage is the same as that of the tab.

Display does not have a well-specified access control policy in today's browsers and standards (corresponding to line 5 in our pseudocode). Our earlier work Gazelle [129] specified an access control policy for display (and Gazelle further advocated that this policy be enforced by the browser kernel, unlike existing browsers). In Gazelle's model, a web site principal delegates its display area to another principal in the form

of cross-domain iframes (or objects, images). Such an iframe (window) is co-owned by both the host page's principal, called landlord, and the nested page's principal, called tenant (both labeled with SOP origins). Principals other than the landlord and the tenant have no access permissions for the window. For the top-level window, the user principal owns it and plays the role of its landlord. Gazelle's policy further specifies how landlord and tenant should access the four attributes of a window, namely the position, dimensions, pixels, and URL location. This specification guarantees that the tenant cannot interfere with the landlord's display, and that the tenant's pixels, DOM objects, and navigation history are private to the tenant. Gazelle's policy is coherent with SOP. In Table 8, we summarized the access control matrix for Gazelle, IE 8, Firefox 3.5, and Chrome 2. The access control of the URL location attribute corresponds to the navigation policy of a browser. Descendant navigation policy allows navigating a descendant window regardless of its origin; this was advocated and implemented over several browsers [32]. Gazelle's policy is *child navigation* policy. (We elaborate in Section 4.1.3.3 that the descendant navigation policy is at conflict with DOM's SOP.) Our tests indicate that Firefox 3.5 and Chrome 2 currently support the child policy, while IE 8 supports the descendant policy. All major browsers allow any window to navigate the top-level window, while Gazelle only allows top-level window navigation from the top-level window's tenant and the user.

XMLHttpRequest allows a web site principal to use scripts to access its document origin's remote data store by issuing an asynchronous or synchronous HTTP request to the remote server [134]. XMLHttpRequest2 [135] and XDomainRequest have been recently proposed and implemented in major browsers to allow cross-origin communications with remote servers, where HTTP authentication data and cookies are not sent by default. These networking capabilities are not shared and strictly belongs to a web site principal labeled with a SOP origin.

PostMessage is a recently proposed client-side cross-origin communication mechanism that is now implemented in all major browsers. This is also a web site principal's capability which is not shared with any other principals.

The last three resources in the non-shared resource table, namely clipboard, browser history, and geolocation, all belong to the user principal, and web applications should not be able to access them directly. However, they are all accessible by scripts through the DOM API, causing problems that we describe in Section 4.1.5.

4.1.3 The interplay of the resources

From the enumeration of the resources and their respective principal or owner definition in the above section, we derived the following problematic pairs of resources, where the two resources interplay and their principal or owner definitions differ: DOM-cookie, cookie-XMLHttpRequest, and DOM-display. We elaborate on these interplays below.

DOM and cookies interplay because scripts are able to create or modify cookies by using the `document.cookie` property in the DOM API.

With no protocol in cookie's principal definition, cookies are vulnerable to information leaks. A cookie intended for a secure HTTPS principal can be passed over HTTP and be exposed to network attackers. This can be prevented by setting the cookie with the "secure" flag. However, a "secure" cookie can still be set by an HTTP response and be accessed by scripts belonging to an HTTP page as long as their domains are the same. Additionally, different services running on different ports of the same domain can access each other's cookies. Moreover, the path protection of cookies becomes ineffective as a script from a different path can access the cookie based on SOP.

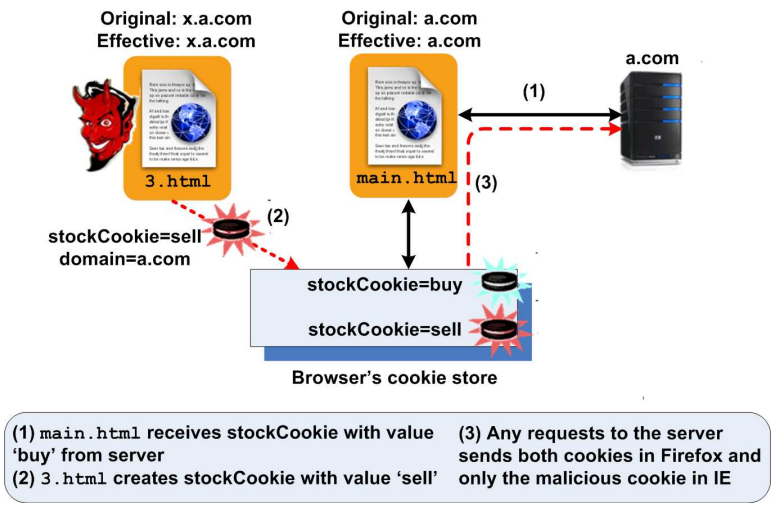


Figure 14: Incoherency arises from the interplay between the access control policies of DOM and cookies

4.1.3.1 DOM and Cookies

The interplay between DOM and cookies also allows the scripts to set the effective domain of a cookie to any suffix of the original domain by setting the *domain* attribute of the cookie. This can lead to inconsistencies in the current browsers. Figure 14 shows a scenario in which such inconsistencies lead to an undefined behavior in the browsers. In this example, a cookie named "stockCookie" with value "buy" is stored in the cookie store for the domain `a.com`. A script injected into a compromised page belonging to `x.a.com` can create another cookie with the same name but with a different value "sell" while setting its domain attribute to `a.com`.

While this leads to a compromised state in the current browsers, different browsers deviate in their behavior creating further inconsistencies in the web applications supporting multiple browsers. Firefox 3 sets this cookie with a domain value of `.a.com` resulting in multiple cookies with the same name in browser's cookie store. The browser attaches both cookies (genuine cookie with domain `a.com` and evil cookie with domain `.a.com`) to any server requests to `a.com`. The server only receives the

cookie's name-value pair without any information about its corresponding domain. This results in the server receiving two cookies with the same name. Since server-side behavior is not defined in case of duplicate cookies [138], it leads to inconsistent state at `a.com`'s server. In case of IE 8, the original cookie value is overwritten and only the wrong cookie value is received by the server.

4.1.3.2 Cookies and XMLHttpRequest

Cookies and XMLHttpRequest interplay because XMLHttpRequest can set cookie values by manipulating HTTP headers through scripts. XMLHttpRequest's owner principal is labeled by the SOP origin, while cookie has a different principal definition (Section 4.1.2).

If a server flags a cookie as "HttpOnly", the browser prevents any script from accessing (both reading and writing) the cookie using the `document.cookie` property. This effectively prevents cookies being leaked to unintended parties via cross-site scripting attacks [93].

The purpose of HttpOnly cookies is that such cookies should not be touched by client-side scripts. However, XMLHttpRequests are created and invoked by client-side JavaScript code, and certain methods of the XMLHttpRequest object facilitate access to cookies: `getResponseHeader` and `getAllResponseHeaders` allow reading of the "Set-cookie" header, and this header includes the value of HttpOnly cookies. Another method, `setRequestHeader`, enables modification of this header to allow writing to HttpOnly cookies.

Some of the latest browsers have tried to resolve this issue with varied success. IE 8 currently prevents both read and write to cookies via "Set-cookie" header, but still allows access via "Set-cookie2" header [73]. Firefox has also recognized and fixed the issue for cookie reads: their fix prevents XMLHttpRequest from accessing cookie

headers of any response, whether or not the HttpOnly flag was set for those cookies [97]. This is a bold step taken by Firefox, as our results show that a considerable number of web pages still read cookie headers from XMLHttpRequest (Section 4.3). However, we have still observed the writing issue with HttpOnly cookies using Firefox 3.5. A script can set a cookie with the same name as the HttpOnly cookie and can have a different value set using the `setRequestHeader` method. This results in a duplicate cookie being sent to the server, thus creating an inconsistent state on the server side.

4.1.3.3 DOM and Display

One incoherence takes place on URL location of a window. The descendant navigation policy (Section 4.1.2) is at conflict with DOM's SOP. Descendant navigation policy allows a landlord to navigate a window, a resource created by its descendant through a DOM API, even if the landlord and the descendant are different principals. This gives a malicious landlord more powerful ways to manipulate a nested, legitimate sites than just overdrawing: with overdrawing, a malicious landlord can imitate a tenant's content, but the landlord cannot send messages to the tenant's backend in the name of the tenant. As an example attack, imagine that an attacker site nests a legitimate trading site as its tenant. The trading site further nests an advisory site and uses a script to interact with the advisory window to issue trades to the trading site backend (e.g., making a particular trade based on the advisory's recommendation shown in the URL fragment). With just one line of JavaScript, the attacker could navigate the advisory window (which is a descendant) and create unintended trades.

Another conflict lies in the access control on the pixels of a window. DOM objects are ultimately rendered into the pixels on the screen. SOP demands non-interference between the DOM objects of different origins. However, existing browsers allow intermingling the landlord's and tenant's pixels by overlaying transparent tenant iframes

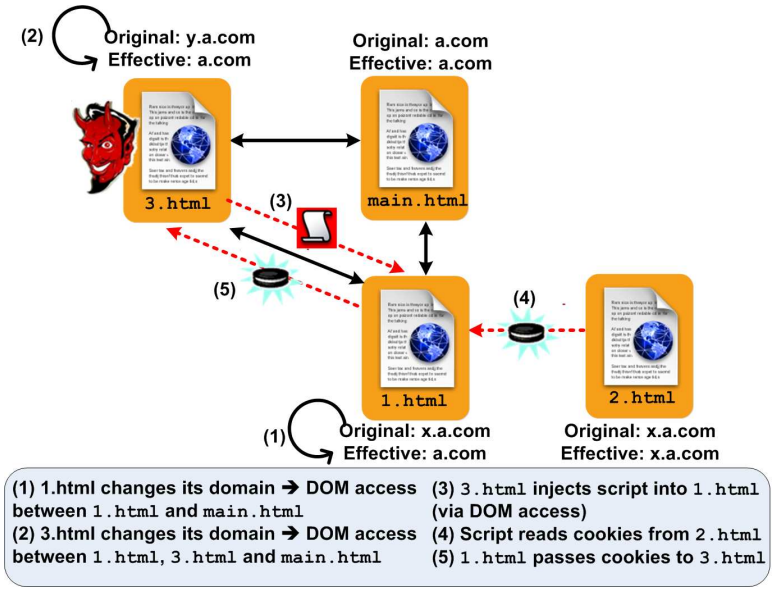


Figure 15: Lack of effective principal ID consideration in cookie's access control policy

on the landlord, deviating from the non-interference goal of SOP. This enables an easy form of clickjacking attacks [46]. In contrast, Gazelle advocates cross-principal pixel isolation in accordance with SOP (Table 8, row "pixels").

4.1.4 Effective Principal ID

Browsers allow cross-principal sharing for "related" sites by allowing sites to change their principal ID via the `document.domain` property [111]. This property can be set to suffixes of a page's domain to allow sharing of pages across frames. For example, a page in one frame from `x.a.com` and a page from `www.a.com` initially cannot communicate with each other due to SOP restrictions. This is one of the few methods for cross-origin frames to communicate before the advent of postMessage [24]. However, changing `document.domain` violates the principle of least privilege: once a subdomain sets its domain to its suffix, there is no control over which other subdomains can access it.

Furthermore, almost no existing access control policies of today's browsers take

such "effective" principal IDs into consideration. In the following subsections, we examine how the disregard of effective principal IDs leads to dual identities and incoherencies exploitable by attackers. In our attack model, an attacker owns a subdomain (through third-party content hosting as in iGoogle or by exploiting a site vulnerability). As we will show in the following sections, the attacker can leverage `document.domain` to penetrate the base domain and its other subdomains.

4.1.4.1 Cookie

Any change of origin using `document.domain` only modifies the effective principal ID for DOM access and does not impact the domain for cookie access. Figure 15 shows an attack to exploit this inconsistent behavior of browser policy design. In this scenario, a page `1.html` in domain `x.a.com` changes it effective domain to `a.com`. As a result, it can access the DOM properties of other pages belonging to `a.com`, but it can no longer access the pages of its original domain `x.a.com`. However, since the effective domain does not change for cookie access, the page still maintains access to the cookies belonging to its original domain. This inconsistent dual identity possessed by the page acts as a bridge to access cookies from both the original domain and the effective domain.

In order to launch the attack, an attacker (after owning a subdomain page) first assumes the identity of `a.com` and subsequently injects a script into the page `1.html`. This injected script can now read and write the cookies belonging to `x.a.com` including any cookies created later. Effectively, if the attacker can compromise a page in one of the subdomains, he can access the cookies of any other subdomains that change their effective origin to the base domain.

4.1.4.2 XMLHttpRequest

Change of origin for scripts does not change the effective principal ID for XMLHttpRequest usage. This enables a (malicious) script in a (compromised) subdomain

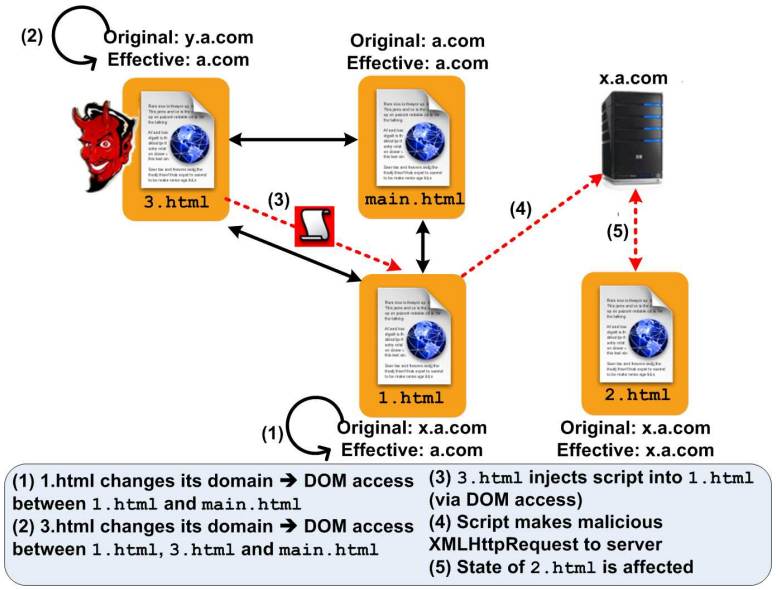


Figure 16: Lack of effective principal ID consideration in XMLHttpRequest's access control policy

to issue XMLHttpRequest to the servers belonging to the base domain and its other subdomains. The attack scenario is illustrated in Figure 16. Page `1.html` has changed its effective domain value to `a.com` from the original value of `x.a.com`. With no effect on XMLHttpRequest usage, scripts in `1.html` can still make requests to the server belonging to `x.a.com`. This again gives a script a dual identity – one for DOM access (`a.com`) and another for XMLHttpRequest (`x.a.com`). Therefore, an attacker compromising any subdomain can inject a script into `1.html` via DOM access, and this script can then make XMLHttpRequest calls to the original domain of the page. Since a well-crafted XMLHttpRequest can change the server-side state for the web application, and this state might be shared between other pages within the domain `x.a.com`, such attack can possibly impact all pages belonging to `x.a.com`.

4.1.4.3 postMessage

`postMessage` also ignores any `document.domain` changes: if `x.a.com` changes domain to `a.com` and sends a message to `y.b.com`, `y.b.com` still sees the message's origin

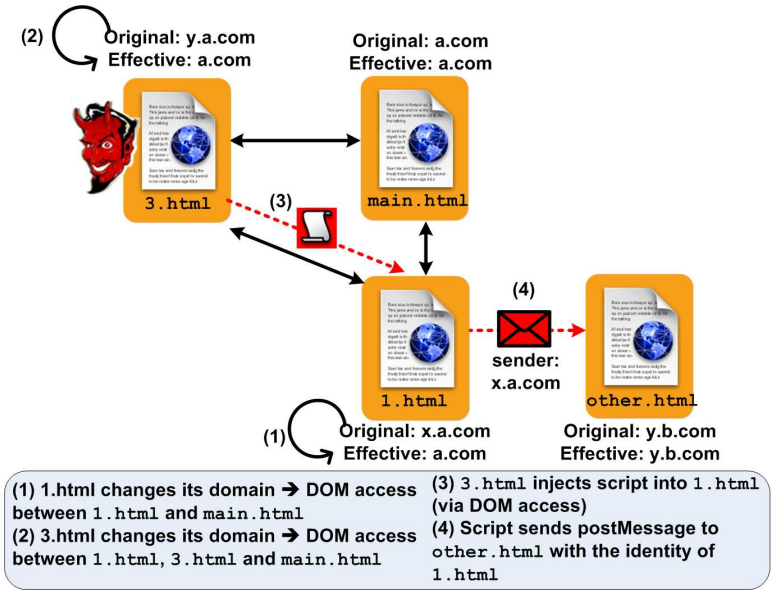


Figure 17: Lack of effective principal ID consideration in postMessage

as `x.a.com`. Also, if `y.b.com` changes its domain to `b.com`, `x.a.com` still has to address messages to `y.b.com` for them to be delivered. This gives the attacker (with a compromised subdomain) an opportunity to send messages while masquerading under the identity of another subdomain (Figure 17).

4.1.4.4 Storage

Based on our tests, IE 8 does not take any `document.domain` changes into consideration for both local storage and session storage. Firefox 3.5 also ignores effective principal ID for local storage. However, for session storage, any domain changes via `document.domain` are considered: the old session storage is lost for the original domain and a new session storage is created for the effective principal.

Inconsistency arises when `document.domain` changes are ignored (for both session storage and local storage in IE; for only local storage in Firefox). An attacker (being able to inject a script into one of the pages of any subdomain, say `x.a.com`) can change its origin to the base domain `a.com` and can successfully inject a script into

the DOM of the base domain or any other origins (e.g., `y.a.com`) that change identity to the base domain. Since access control checks on storage rely on original domain (i.e., `y.a.com`), the malicious script can now freely access the storage belonging to `y.a.com`.

4.1.5 The User Principal

In this work, we introduce the concept of the *user principal* in the browser setting. The user principal represents the user of the browser. Unfortunately, it has often been neglected in browser access control policies.

While a web application does manage the user's data and experience for that particular application (e.g., a user's banking data at a banking site), certain browser resources or data belong to the user exclusively and should not be accessible by any web site without user permissions. Such resources include: user's private data, such as clipboard data and geolocation; user actions, such as clicking on the forward and back button; devices, such as camera and microphone; and browser UI, including the current tab window (top-level window).

Unfortunately, in today's browsers, some of these resources are directly exposed to web applications through the DOM API. This breaks the fundamental rule of protecting resources belonging to different principals from one another, as the user principal's resources can be accessed and manipulated by site principals. This can result in privacy compromises, information leaks, and attacks that trick users into performing unintended actions. In this section, we examine the user principal resources and describe our findings on how they may be accessed improperly by web applications.

4.1.5.1 User actions

The `focus` and `blur` properties of the `window` object allow web sites to change focus between the windows that they opened irrespective of the origins. This enables an

attacker site to steal focus or cause the user to act on a window unintentionally.

The `window` object has a `history` property with an array of user-visited URLs. Browsers have been denying any site's access to this array to protect user privacy, but they do allow a site to navigate the browser back and forward in history through the `back()` and `forward()` methods [60]. Worse, our tests indicate that Firefox 3 and Google Chrome 2 allow any child window to navigate the top-level window back or forward in history *irrespective* of the origin. In many cases this is just a nuisance, but some properly-crafted history navigation by a malicious application can lead to more severe damage. For example, the user might be tricked to make multiple purchases of the same product.

We have also investigated synthetic event creation. The DOM API allows a site to generate synthetic mouse or keyboard events through the `document.createEvent()` method (or `document.createEventObject()` in IE). In IE, a programmer could directly invoke a `click()` method on any HTML element to simulate user clicks. These techniques are useful for debugging purposes. To our delight, all major browsers are careful not to let a web site to manipulate another site's user experience with these synthetic user events. Note that it is benign for a site to simulate the user's actions for itself, since loading and rendering site content can by itself achieve any effects of simulating user actions (e.g., simulating a mouse click is equivalent of calling the `onclick` function on the corresponding element).

4.1.5.2 Browser UI

An important part of the browser UI is the current tab window, or top-level window. In today's browsers, any web site loaded in any window is able to reposition and resize a top-level window through the `moveTo`, `moveBy`, `resizeTo`, and `resizeBy` properties of the top-level window. Resizing the currently active top-level window effectively resizes the browser window. Firefox 3 allows an application to resize a

browser window even in the presence of multiple tabs, while IE 8 and Chrome 2 do not allow this. A site can also open and close a top-level window using `open` and `close` methods. The use of `open` method has been mitigated through built-in popup blockers. IE 8 allows any frame to close a top-level window irrespective of the origin, while Firefox 3 and Chrome 2 prevent this from happening. These capabilities allow an attacker site (even when deeply nested in the DOM hierarchy, say a malicious ad) to directly interfere with the user's experience with the browser UI.

Some of the other loopholes in browser UI have already been fixed. For example, the status bar can no longer be set by a web site.

4.1.5.3 User-private state

Jackson et al. have shown that a user's browsing history can be exposed by inspecting the color of a visited hyperlink [78], raising privacy concerns. The hyperlink's color is intended for the user, and it is not necessary for web sites to be able to read it.

The clipboard data also belongs exclusively to the user principal. All versions of IE since 5.0 support APIs to access clipboard data. A web site can get contents of a user's clipboard by successfully calling `window.clipboardData.getData("Text")`. Depending on the default Internet security settings, the browser may prompt user before getting the data. However, the prompt does not identify the principal making the request (simply using the term "this site"). As a result, a malicious script embedded on a third-party frame may trick the user into giving away his clipboard because he thinks that such access is being requested by the trusted top-level site.

Geolocation is one of the latest browser features that allows a site to determine the client's location by using the `navigator.geolocation` [72] interface. At the time of writing, Firefox 3.5 is the only stable production browser supporting this HTML5 feature. Geolocation is user-private data. Today's browsers do ask user permission before accessing it. However, issues arise when a site embeds content from multiple

principals (i.e., in frames), and more than one origin needs access to geolocation information. The geolocation dialog is active for only one origin at a time; if there is a request to access geolocation from `b.com` while the dialog for `a.com` is still active, it is ignored — the principal that succeeds in invoking the geolocation dialog first wins. Therefore, if a malicious party manages to embed a script (or a frame) on the page, it can prevent the main site from triggering the geolocation dialog by invoking the dialog first. As a result, the malicious party can create denial-of-service against the main site, preventing it from retrieving a user's geolocation information. Additionally, it could trick the user into giving away location to itself rather than the main site (e.g., using phishing domain names like `www.gooogle.com`).

Changing `document.domain` also generates inconsistencies. The geolocation prompt is designed to work only with the original principals, and even if a site changes identity, the prompt still displays the original domain as the requesting domain. For an example site `good.a.com` that changes its document.domain to `a.com`, this causes the following problems:

- If an attacker site `evil.a.com` changes its document.domain to `a.com`, it can steal position information from `good.a.com`, if `good.a.com` has stored or displayed this information in a place that is accessible via the DOM (e.g., using `parent.document.getElementById( "coords").innerHTML`).

- If another site `evil.a.com` also changes its domain to `a.com`, it could impersonate `good.a.com`, by using `parent.navigator.geolocation .getCurrentPosition`, which would trigger the access prompt using `good.a.com`, instead of `evil.a.com`.

4.2 The WebAnalyzer Measurement Framework

To achieve consistent browser access control policies, browser vendors need to remove or modify the features that contribute to incoherencies. For example, disallowing domain-setting for cookies, eliminating `document.domain`, and removing support for

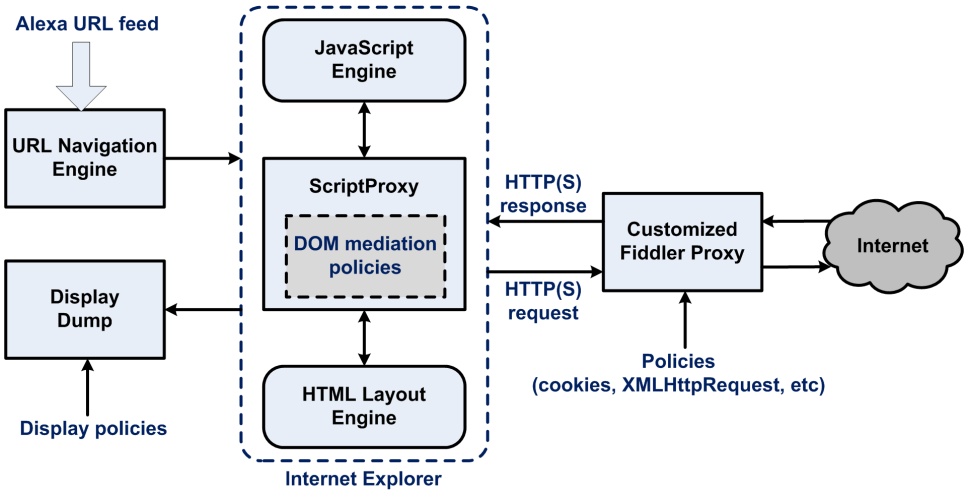


Figure 18: High-Level Architecture of IE_{WA}.

accessing user principal resources are steps towards secure new browsers. However, this begs the question of what the cost of these feature removals is and how many web sites will break as a result. In today's highly competitive browser market, backward compatibility with the existing web is paramount.

To help browser vendors balance security and compatibility, we set off to build a measurement system to measure the cost of security. Many previous web compatibility studies have been *browser-centric*: they have evaluated the degree to which a given browser supports various web standards or is vulnerable to attacks [74, 139]. In contrast, we take a *web-centric* perspective and actively crawl the web to look for prevalence of unsafe browser features on existing web pages. Compared to existing crawlers, however, static web page inspection is insufficient. Dynamic features such as AJAX or post-render script events require us to actively render a web page to analyze its behavior at run time. Moreover, the incoherencies we identified in Section 4.1 require analysis of not just a page's JavaScript execution [137], but also DOM interactions, display layout, and protocol-layer data.

To address these challenges, we have constructed a scalable, execution-based crawling platform, called WebAnalyzer, that can inspect a large number of web pages

by rendering them in an instrumented browser. The platform consumes a list of URLs (defined by a human operator or generated by a traditional web crawler), and distributes them among virtual machine workers, which renders them using IE_{WA}, a specially instrumented version of Internet Explorer. IE_{WA} provides dynamic mediation for all browser resources, and detects when a resource invocation matches one of preset policy rules. Even though our framework is extensible to a large variety of browser policies, we concentrate on "unsafe feature" rules derived from our analysis in Section 4.1.

To build IE_{WA}, the central piece of our measurement platform, we leverage public COM interfaces and extensibility APIs exported by Internet Explorer 8. Figure 18 shows the architecture of IE_{WA}, which centers around three major interposition modules: (1) a script engine proxy, which provides JavaScript and DOM interposition, (2) a network proxy based on Fiddler [84], and (3) display dumper, which enables custom analysis of a page's layout as it is visible to the user. Next, we discuss each module in turn.

Script engine proxy. We build on our earlier system in MashupOS [128] to implement a JavaScript engine proxy (called *script engine proxy* (SEP)): SEP is installed between IE's rendering and script engines, and it mediates and customizes DOM object interactions. SEP exports the script engine API to IE's renderer, and it exports the DOM and rendering interfaces to IE's script engine. Each DOM object is interposed by a corresponding object wrapper. When IE's script engine asks for a DOM object from the rendering engine, SEP intercepts the request, retrieves the corresponding DOM object, associates the DOM object with its wrapper object inside SEP, and then passes the wrapper object back to the original script engine. Any subsequent invocation of wrapper object methods from the original script engine passes through SEP. SEP is implemented as a COM object and is installed into IE by modifying IE's JavaScript engine ID in the Windows registry.

Network interposition. In addition to SEP, we route the browser's network traffic through a proxy to monitor all HTTP/HTTPS requests and analyze cookie transfers as well as network APIs like XMLHttpRequest. Our network proxy is implemented using the *FiddlerCore* interfaces provided by the public-domain Fiddler web debugging proxy [57, 84].

Display analysis. In order to evaluate display policies, it is necessary to analyze a browser's visual output as seen by the user. For this purpose, we use a customized version of IE's rendering engine that exposes COM interfaces to extract a textual representation of a particular page's visual layout at any stage of rendering. In our current evaluation, we use these COM interfaces to save a snapshot log of IE's display after a page has fully loaded. Because some pages have post-render events that alter layout, we wait an additional 5 seconds before taking a display snapshot. Snapshot logs provide a mapping between a page's objects and their layout properties, such as position, dimensions, or transparency. They can be analyzed offline for the presence of unsafe frame overlapping behavior or other dangerous page layouts.

Navigation. To facilitate automatic analysis for a large number of URLs, IE_{WA} includes a URL navigation engine, which utilizes IE's extensibility interfaces, such as `IWebBrowser2`, to completely automate the browser's navigation. In addition to pointing the browser to new URLs, this module also cleans up state such as pop-ups between consecutive URLs, detects when sites fail to render (e.g., 404 errors), and recovers from any browser crashes.

Visiting a site's home page is sometimes insufficient to invoke the site's core functionality. For example, a feature may be accessed only when the user clicks on a link, types search queries, or causes mouse event handlers to run.

It is difficult and time-consuming to fully automate a site's analysis to study all possible features and pages that could be invoked using all combinations of user input. Instead of aiming for complete coverage within a particular site, we enhanced

our navigation engine with simple heuristics that simulate some user interaction. After rendering a site's home page, IE_{WA} will find and simulate a click on at most five random links, producing five random navigation events. In addition, IE_{WA} will check for presence of a search form, fill it with random keywords, and submit it. We restrict all simulated navigations to stay within the same origin as a site's home page.

These simple enhancements maintain our ability to examine a large number of sites while adding the ability to properly handle many (but not all) sites with home pages that do not invoke the site's main functionality. For example, we can navigate to a random article on Wikipedia, a random video on YouTube, a random profile on MySpace, a random Twitter feed, and a random search query on Google. We evaluate the success of this methodology against a user-driven browsing study in Section 4.3.7 and discuss its limitations in Section 4.4.

Performance. We deployed our system on several desktop machines, each with an Intel 2.4 GHz quad-core CPU and 4 GB of RAM. Our IE_{WA} workers run inside a Windows Vista VMware virtual machine to prevent malware infection. We executed multiple workers in each VM, isolating them from one another using different UIDs and different remote desktop sessions.

On such a setup, one IE_{WA} worker is able to analyze about 115 typical web sites per hour. Each site's processing time includes the home page, five random link clicks, and one form submission, as well as overheads introduced by IE_{WA}'s three interposition modules. We found that we could execute up to eight parallel workers in one VM, for a throughput of 900 sites per VM, before saturating the CPU. Optimizing this infrastructure for performance was not a goal of this thesis and is left as future work.

4.3 Experimental Results

Our analysis in Section 4.1 provides an understanding of the security characteristics of the current access control policies in browsers. In this section, we complete the

other half of the equilibrium by using the measurement infrastructure presented in Section 4.2 to study the prevalence of unsafe browser features (analyzed in Section 4.1) on a large set of popular web sites. By presenting both sides, we enable the browser vendors to make more informed decisions about whether or not to continue supporting a particular unsafe feature based on its real-world usage.

4.3.1 Experimental overview

4.3.1.1 Choosing the sites for analysis

Instead of randomly crawling the web and looking for unsafe features, we decided to focus our attention on the "interesting" parts of the web that people tend to visit often. Accordingly, to seed our analysis, we take the set of 100,000 most popular web sites ranked by *Alexa* [27], as seen on November 9, 2009, as our representative data set. The data collection and analysis were completed in the last week of February 2010.

4.3.1.2 Defining the compatibility cost

We define the cost of removing a feature to be the number of Alexa-ranked, top 100,000 sites that use the feature.

We conservatively assume that disallowing a feature will significantly hinder a site's functionality, whereas it could simply cause a visual nuisance. A more detailed analysis on the effect of policy changes on page behavior is promising but is left as future work.

4.3.1.3 High-level results

We obtained our results by rendering each of the 100,000 seed links using WebAnalyzer, saving all interposition logs for offline analysis. This way, we were able to obtain data for 89,222 of the 100,000 sites. There are several reasons why no data was produced for the rest of sites. First, some sites could not be accessed at the time

Table 9: Usage of various browser features on popular web sites (February 2010). Analysis includes 89,222 sites.

Measurement Criteria	Total instances (count)	Unique sites	
		Count	Percentage
document.cookie (read)	5656310	72587	81.36%
document.cookie (write)	2313359	68230	76.47%
document.cookie domain usage (read)	2032522	59631	66.83%
document.cookie domain usage (write)	1226800	41327	46.32%
Secure cookies over HTTP	259	62	0.07%
Non-secure cookies over HTTPS	15589	4893	5.48%
Use of "HttpOnly" cookies	33180	14474	16.22%
Frequency of duplicate cookies	159755	4955	5.55%
Use of XMLHttpRequest	19717	4631	5.2%
Cookie read in response of XMLHttpRequest	1261	265	0.30%
Cross-origin descendant navigation (reading descendant's location)	6043	61	0.07%
Cross-origin descendant navigation (changing descendant's location)	0	0	0.00%
Child navigation (parent navigating direct child)	22572	6874	7.7%
document.domain (read)	1253274	63602	71.29%
document.domain (write)	8640	1693	1.90%
Use of cookies after change of effective domain	295960	1569	1.76%
Use of XMLHttpRequest after change of effective domain	225	87	0.10%
Use of postMessage after change of effective domain	0	0	0.00%
Use of localStorage after change of effective domain	42	10	0.01%
Use of local storage	1227	169	0.19%
Use of session storage	0	0	0.00%
Use of fragment identifier for communication	5192	3386	3.80%
Use of postMessage	6523	845	0.95%
Use of postMessage (with no specified target)	0	0	0.00%
Use of XDomainRequest	527	125	0.14%
Presence of JavaScript within CSS	224266	4508	5.05%

of our analysis due to failed DNS lookups, "404 Not Found" errors, and other similar access problems. Second, some sites timed out within our chosen threshold interval of 2 minutes, due to their slow or continuous rendering. We decided to drop any such sites from our analysis. Finally, some sites did not contain any JavaScript code, and as a result they did not trigger our event filters. Nonetheless, we believe that we have been able to analyze a sufficiently large set of sites with a reasonable success ratio, and our data set and the scope of measurement is much larger than that used by earlier related studies [137].

Tables 9, 10, and 11 present the results of our analysis, showing how frequently each feature we analyzed earlier is encountered. Next, we organize our findings according to our discussion in Section 4.1 and discuss their implications on future browser security policies.

4.3.2 The interplay of browser resources

4.3.2.1 DOM and Cookies

Cookie usage is extremely popular, and so is their programmatic DOM access via `document.cookie`, which we found on 81% web sites for reading and 76% of web sites for writing cookie values, respectively. The use of the cookie's `domain` attribute is also widespread (67% of sites), with about 46% of sites using it to actually change the domain value of the cookie. As a result, the issues described in Section 4.1.3.1 cannot be solved by simply deprecating the usage of this attribute and changing the principal definition of cookies. One possible approach to solve the inconsistency issue with cookie handling is to tag the cookie with the origin of the page setting the cookie. This information should be passed to the server to allow the server to differentiate between duplicate cookies.

Section 4.1.3.1 also identified inconsistencies pertaining to cookies and HTTP/HTTPS, which we now support with measurements. First, 0.07% of sites alarmingly send `secure` cookies over HTTP. This effectively tampers with the integrity of cookies

that may have been intended for HTTPS sessions [31]. Fortunately, it appears that this functionality can be disallowed with little cost. Surprisingly, a much larger number of sites (5.48%) sent HTTP cookies over HTTPS. The HTTP cookies cannot be kept confidential and are accessible to HTTP sessions. Our recommended solution to this problem is that the "secure" flag should be enforced for any cookies passed over an HTTPS connection even if the web developer fails to set the flag. This would still enable the HTTPS site to access the cookie for its own functionality and any sharing with the HTTP site should be done explicitly.

We found a large number of sites (16.2%) using HttpOnly cookies, which is an encouraging sign — many sites appear to be tightening up their cookie usage to better resist XSS attacks.

4.3.2.2 Cookies and XMLHttpRequest

Our measurements show that the issues arising from undesirable interplay of `XMLHttpRequest` and HttpOnly cookies (Section 4.1.3.2) can possibly be eliminated, since very few sites (0.30%) manipulate cookie headers in XMLHttpRequest responses.

4.3.2.3 DOM and Display

Section 4.1.3.3 argued that the descendant navigation policy is at conflict with SOP for DOM. We observe iframe navigations on 7.7% of sites and all of them are child navigation (regardless of the origin). The absence of descendant navigation in the top 100,000 sites indicates a potentially very low cost to remove it.

In addition, we have analyzed the visual layouts of all sites to determine whether there are dangerous pixel interplays between windows of different principals (Section 4.1.3.3). Our results are summarized in Table 10[1]. We found that 41% of sites

[1]Our display analysis was performed in December 2009, separately from script engine and network analysis that we performed in February 2010, causing a slight difference in the number of successfully rendered sites in Tables 10 and 9.

Table 10: Summary of display layouts observed for the top 100,000 Alexa web sites (December 2009). 89,483 sites were rendered successfully and are included in this analysis.

Sites containing at least one `<iframe>`	36549 (40.8%)
Average number of `<iframe>`'s per site	3.2
Sites with at least one pair of overlapping frames	5544 (6.2%)
Sites with at least one pair of overlapping cross-origin frames	3786 (4.2%)
Sites with at least one pair of *transparent* overlapping frames	1616 (1.8%)
Sites with at least one pair of *transparent* overlapping cross-origin frames	1085 (1.2%)

Table 11: Prevalence of resources belonging to the user principal on popular web sites. Analysis includes 89,222 sites.

Measurement Criteria	**Total instances (count)**	**Unique sites**	
		Count	**Percentage**
Setting top-level window's location	55759	2851	3.20%
Change focus of window	5221	2314	2.59%
Reading color of hyperlinks	82587	1560	1.75%
Accessing browser's history	1910	721	0.81%
Use of `defaultStatus` (write)	1576	241	0.27%
Reading user's Geolocation	251	149	0.17%
Use of `resizeTo`	339	134	0.15%
Use of `defaultStatus` (read)	528	108	0.12%
Use of `moveTo`	258	100	0.11%
Close a window	130	86	0.10%
Access to user's clipboard	24	17	0.02%
Blur a window	54	13	0.01%
Use of `resizeBy`	13	8	0.01%
Use of `moveBy`	4	1	0.00%
Use of `outerWidth`	2	1	0.00%
Use of `outerHeight`	4	1	0.00%

embed at least one `iframe`, and the average number of iframes embedded on a particular page is 3.2. Overlapping iframes appear to be common — 6.2% of sites contained at least one overlapping pair of iframes — but only 29% of these overlaps involved transparent iframes. Most (68%) overlapping scenarios involve different principals.

The most dangerous situations occur when a transparent frame is overlaid on top of a frame belonging to a different principal (Section 4.1.3.3). We identified 1,085 sites (1.2%) that contained at least one pair of *transparent*, cross-origin overlapping iframes. We observed that most of these overlaps involved domains serving ad banners, so the main site functionality might remain unaffected if the dangerous transparency is disallowed.

Summary. We found that interplays between DOM and cookies have a high compatibility impact, while removing the interplays between cookies and XMLHttpRequest would affect only 0.30% of sites. For interplays related to display, we found that descendant navigation can be disallowed with no cost, while disallowing overlaps between transparent cross-origin frames would affect 1.2% of sites.

4.3.3 Changing effective Principal ID

In Section 4.1.4, we showed that `document.domain` is an unsafe and undesirable part of today's web, as observed by others as well [138]. Unfortunately, we found its usage on the web to be significant: 1.9% of sites change their effective domain via `document.domain`.

We mentioned certain features which become incoherent when combined with `document.domain`. Cookies are accessed by about 1.76% of the sites after a change in effective domain, making it difficult to enforce a unified effective domain for cookie access (Section 4.1.4.1). Only 0.08% of sites use XMLHttpRequest after an effective UID change (Section 4.1.4.2), so it appears possible to make XMLHttpRequest respect effective domain with little cost. The same holds true for postMessage — we found no sites using postMessage after an effective UID change. The new local storage abstractions are not widespread — only 0.19% of the sites were using `localStorage` (0.01% after an effective domain change), and no sites were using `sessionStorage` — so we anticipate that origin-changing weaknesses that we outlined in Section 4.1.4.4

can be removed with little compatibility cost.

Summary. Overall, while disallowing `document.domain` completely carries a substantial cost (1.9% of sites), browsers can eliminate its impact on XMLHttpRequest, local storage, and postMessage at a much lower cost (0.19% of sites total). On the flip side, browser vendors have to make a much tougher choice (affecting 1.76% of sites) to prevent effective UID inconsistencies pertaining to cookies.

4.3.4 Resources belonging to the user principal

Table 11 shows the results of our analysis for the cost of protecting user-owned resources discussed in Section 4.1.5. The cost of tightening access control for user resources appears to be low with the exceptions of link-color access (1.8%), the focus-changing functions (2.6%), and setting top-level window location (3.2%).

Interestingly, 149 sites (0.17%) already use the new Geolocation primitives [72]. This number seems low enough for browsers to take actions to tighten its access control.

Overall, we found that 12 of the 16 user-principal APIs we examined can be removed while collectively affecting only 0.80% of unique sites.

4.3.5 Other noteworthy measurements

We measured prevalence of some primitives for cross-frame and cross-window communication, which are critical for cross-principal security. Fragment identifier messaging is most popular, being found at 3.8% of sites. A non-negligible number (0.95%) of sites have already adopted `postMessage`, and all sites use its newer definition that requires specifying the target window [32]. Another safer alternative for cross-domain communication, XDomainRequest, is also being slowly adopted (0.14%).

Using JavaScript within CSS has long been considered dangerous [138]. We found this pattern in use on about 5% of the sites.

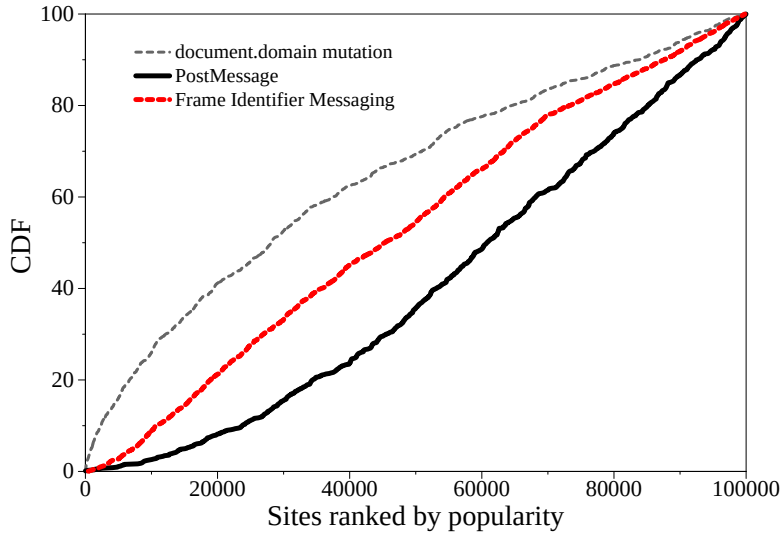


Figure 19: A CDF for prevalence of cross-frame communication mechanisms according to the ranking of sites that use them.

4.3.6 Correlating unsafe features and site popularity

Next, we consider how the popularity of sites correlates with prevalence of unsafe features. A policy is more costly to correct if it is used by very highly ranked sites, since more people would visit them and encounter broken functionality. Fortunately, we found that most features do not exhibit a significant popularity bias, behaving uniformly with no regard to a site's popularity. Nevertheless, we found some exceptions. Figure 19 shows a CDF of the usage of various mechanisms that could be used for cross-frame communication according to the sites' ranking. Interestingly, fragment identifier messaging has little dependence on popularity, `document.domain` tends to be used more by higher-ranked sites, and `postMessage` is found more on lower-ranked sites, with very little use in the top 2000 sites. This went against our hypothesis that higher-ranked, high-profile sites would likely be written using the latest and safest web standards. A possible explanation could be that the top sites are motivated to use features compatible with the largest number of browsers and client platforms.

As another example, Figure 20 diagrams the prevalence of resources belonging

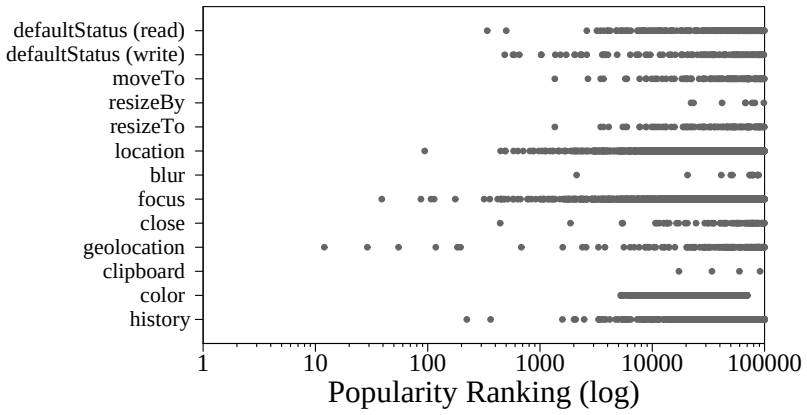


Figure 20: A CDF for prevalence of user-owned resources according to the ranking of sites that use them.

to the user principal according to the ranking of the sites that use them (a dot is displayed for every site using a particular feature). Some features, such as `resizeBy` or clipboard access, are only found on very low-ranked sites and are thus good candidates to remove with little impact. Only a handful of features appear in the top 100 sites, where compatibility cost is very high for any site.

4.3.7 Methodology validation using user-driven analysis

In the previous sections, we examined sites by visiting their home pages and relying on WebAnalyzer's heuristics (see Section 4.2) to simulate a few basic user actions to invoke additional functionality that may be hidden behind "splash" home pages. However, our methodology may miss site functionality that requires user login forms (e.g., on Facebook), other more sophisticated user event handlers (e.g., mouse movements), or following many links away from the home page. In general, it is very difficult, if not impossible, to simulate user actions that open access to representative features of an arbitrary site.

To evaluate the limitations of our heuristics-driven approach, we conducted a user-driven examination of the top 100 Alexa sites. To do this, one of the authors manually visited these sites with IE_{WA} and used his best judgement to invoke the site's

Table 12: Comparison of user-driven analysis vs. WebAnalyzer for the top 100 Alexa sites. Features not shown here were used by zero sites for both user-driven and WebAnalyzer studies.

Measurement Criteria	Number of sites	
	WebAnalyzer	**Manual**
document.cookie (read)	93	86
document.cookie (write)	86	76
document.cookie domain usage (read)	78	70
document.cookie domain usage (write)	59	59
Secure cookies over HTTP	0	2
Non-secure cookies over HTTPS	11	8
Use of "HttpOnly" cookies	27	30
Frequency of duplicate cookies	17	8
Use of XMLHttpRequest	32	28
Cookie read in response of XMLHttpRequest	0	0
Cross-origin descendant-navigation (reading descendant's location)	0	0
Cross-origin descendant-navigation (changing descendant's location)	0	0
Child navigation (parent navigating direct child)	1	2
document.domain (read)	78	59
document.domain (write)	18	19
Use of cookies after change of effective domain	18	19
Use of XMLHttpRequest after change of effective domain	4	2
Use of localStorage after change of effective domain	2	1
Use of session storage	0	0
Use of local storage	4	3
Use of fragment identifier for communication	0	1
Use of postMessage	1	1
Use of XDomainRequest	1	2
Presence of JavaScript within CSS	16	27
Setting top-level window's location	1	2
Change focus of window	2	2
Reading user's Geolocation	3	9

representative functionality. For example, for analyzing Facebook, the author logged into his Facebook account, browsed through several profiles, and invoked several applications such as photo viewing or messaging.

We then compared the results obtained through this manual analysis to those

obtained using WebAnalyzer for the same sites. Table 12 summarizes the results of our comparison. We observe that the numbers of sites using a particular feature are mostly comparable, providing confidence that our heuristic-driven navigation engine in WebAnalyzer works well in practice. Some features have higher prevalence with the user-driven analysis, as expected, but there are only a couple of outliers. For example, Geolocation was found on nine sites, all found on multilingual versions of `maps.google.com`. In manual analysis, the user invoked maps on each of the nine versions of the Google site, where WebAnalyzer randomly picked and followed the link to `Maps` on three of these sites. On the other hand, on several occasions, WebAnalyzer also found features that were missed by manual analysis, as can be seen in higher prevalence for features like reading `document.domain`. This can happen when WebAnalyzer navigates to a link that the user did not examine as part of representative functionality on a given site. Overall, we felt our heuristics-driven approach achieved good coverage, though larger-scale user-driven measurements would still be very valuable in complementing WebAnalyzer measurements.

4.4 Discussion and limitations

Benefits of heuristics-driven automated crawling. In our original design, WebAnalyzer visited only the top-level page of each site we studied. We quickly realized that this analysis failed for sites that hide much of their functionality behind "splash" home pages. This became most apparent when studying the original results for Table 12. We observed that for many sites, clicking on a link or filling out a search form on the home page would expose a noticeably larger (though still not complete) set of functionality. Thus, we augmented WebAnalyzer with simple heuristics that imitate this user behavior (see Section 4.2).

As an example, our original system saw XMLHttpRequest calls on only 13 pages of the top 100 pages, whereas the new one identified 32 such pages (see Table 12).

One of the reasons is that many search sites use XMLHttpRequest to auto-complete the search string as users type it; our old system did not trigger this behavior, whereas our new system triggered it when auto-filling the search textbox. Many other features showed a similarly dramatic jump in prevalence.

Limits of automated crawler-based measurements. Although we believe that our resulting measurements provide a good representation of the use of browser features on popular web sites, it is likely that we missed certain features because the code path to invoke them was not triggered in our analysis. For example, sites like Facebook or banks require a user to sign in, game sites require particular mouse gestures to invoke certain behavior, and numerous sites require appropriate text (such as stock symbols or user's personal data) to be entered into forms. Even if we could solve some of these problems, for example by enumerating all events registered on a page or using a database of dummy usernames and passwords [5], automatically invoking certain features, such as buying products on shopping sites, is inappropriate. This ultimately limits our ability to explore *all* features invoked on today's web.

We also did not try to exhaustively crawl each site. Even in our user-driven analysis (Section 4.3.7), we did not attempt to enumerate and invoke all gadgets on every page of each site. Thus, the results we collect for a particular site cannot be used as a list of *all* features the site might have. Our aim was to favor breadth over depth and obtain good coverage for the representative features of 100,000 sites we tested. While our infrastructure could also be used for exhaustively crawling each site, we would need to dramatically scale up our current infrastructure to cover a comparable number of sites, and we leave this as future work.

Picking the right browser. Some sites check the client's browser version (using the user-agent header) before deciding to invoke a particular code path. Although not a base requirement, we developed WebAnalyzer with IE as the underlying browser. This could prevent code invocations that are intended for non-IE browsers, thereby

leading to missed features. For example, XMLHttpRequest2 [135] is currently not supported by IE, and it would be missed by WebAnalyzer if the site invokes it only after verifying browser support.

A related problem is fallback code that invokes an alternative implementation of a feature that a browser doesn't support. For example, a site could first check whether the browser supports `postMessage` for cross-frame communication, and fall back on fragment identifier messaging if it does not. Because we use IE 8, we will log that this site uses `postMessage`, but older browsers would utilize fragment identifier messaging.

The compatibility cost of features invoked in browser-dependent code paths depends not only on the number of web sites using a feature, but also on the number of visitors utilizing a particular browser that relies on such code. Evaluating the second part of this cost is orthogonal to our goals in this thesis: rather than exploring prevalence of features on web sites, it asks how many of a web site's clients rely on a particular browser. Web server operators can easily answer this question by profiling "user-agent" strings in incoming HTTP requests. As future work, we can integrate other browsers into WebAnalyzer, or we can modify IE_{WA} to render a site with a set of user-agent strings representing other browsers; this would capture a more complete set of the site's code.

Studying other web segments. Our focus on the top 100,000 sites represents a particular segment of the web with a good balance of the very top sites and some of the less popular "tail". However, this still covers only a tiny fraction of the billions of pages on today's web. In addition, our analysis excluded intranet sites, which are hidden from traditional crawlers, and which can influence backwards compatibility decisions for a browser. We leave exploration of these other segments of the web as important future work.

4.5 Summary

In this chapter, we have examined the current state of browser access control policies and analyzed the incoherencies that arise when browsers mishandle their principals by (1) inconsistently labeling resources with principal IDs, (2) inappropriately handling principal identity changes via `document.domain`, and (3) neglecting access control for certain resources belonging to the user principal. In addition to pointing out these incoherencies, we have developed a web compatibility analysis infrastructure and measured the cost of removing many unsafe policies we identified for a large set of popular web sites. Overall, this work contributes to the community's understanding of browser access control policies, and it provides the much-needed answer to the browsers' *compatibility vs. security* dilemma by identifying unsafe policies that can be removed with little compatibility cost.

CHAPTER V

END-TO-END CONTENT INTEGRITY POLICIES

The same-origin policy [111] (SOP) is the key access control policy for the web and browsers. This policy has essentially defined a principal model where web sites are mutually distrusting principals [128,129], and where one site's script cannot access another site's content. However, the authenticity of the principal and the integrity of its content are often at question since much of the web is delivered over HTTP rather than HTTPS. Consequently, network attackers can carry out man-in-the-middle attacks and undermine browsers' access control, even if browsers flawlessly implement the enforcement of the same-origin policy. Such attacks are highly practical today with the prevalence of wireless hotspots and insecurity in the DNS infrastructure [61]. The web requires *end-to-end security* to allow meaningful SOP enforcement in browsers.

HTTPS [109] has the potential to prevent network attacks, but its universal adoption is hindered by its uncacheability at intermediate servers, such as content distribution network (CDN) servers and HTTP proxies, and its performance cost.

Web caching offers significant benefits to web sites and users. It enables web sites to save bandwidth costs and reduce latency for users by outsourcing infrastructure to CDNs and offloading requests to CDN servers. Although CDNs do offer services for HTTPS content [26], this is at the cost of trusting CDN servers to be man-in-the-middle and losing end-to-end security. Furthermore, such services come with a hefty charge of up to $3,000 per month plus bandwidth costs [50]. Web cache proxies can also deliver web content significantly faster to large user communities behind gateways or firewalls, such as mobile users. HTTPS content cannot take advantage of these proxies at all today. We observe that much of the web is cacheable (Section 5.3.1), and

we expect significant growth in cacheable web content as rich media proliferates [6]. To achieve an end-to-end secure web, HTTPS is definitely not the complete answer.

In terms of performance, although GMail has recently demonstrated the ability of serving HTTPS content with low overhead using commodity hardware (1% CPU load, less than 10KB of memory per connection and less than 2% network overhead) [83], a general applicability of their solution to other SSL setups is not clear [8]. Due to differences in HTTPS deployments, it might not be trivial for other web sites to replicate Gmail's performance improvements. Even if the SSL's server overhead is successfully reduced, it still suffers from lack of in-network caching, thus limiting the performance benefits for the clients.

Fortunately, end-to-end security, cacheability, and performance are not at conflict inherently. End-to-end security encompasses (1) end-to-end authentication (i.e., content comes from the right origin[1]) (2) end-to-end content integrity (i.e., content is not tampered), and (3) end-to-end content confidentiality (i.e., content is kept private). For the browser platform to meaningfully enforce its access control policy, both authentication and integrity are needed, but confidentiality is *not* required. Without confidentiality, the content is cacheable at intermediate web servers. HTTPS provides all three properties simultaneously and is hence not cacheable.

In this work, we propose *HTTPi* as a protocol to support only end-to-end authentication and content integrity. We advocate that web sites use HTTPS for requests that require end-to-end confidentiality, and HTTPi for all other requests.

HTTPi revives the signature mode of operation from SHTTP [110], which was a proposal that unsuccessfully competed with SSL and HTTPS. In our work, we give a practical and comprehensive design and implementation of such a content-signature-based protocol. While HTTPi requires both browser and server-side modifications, our design does not require changes at intermediate nodes, such as proxies, for caching

[1]Client authentication is at the discretion of web sites.

HTTPi content (Section 5.1.1). Our design also ensures that progressive content loading in browsers is not hindered by HTTPi, and that this incurs minimal overhead in both computation and bandwidth (Section 5.1.1). Because signatures can be computed offline and cached for static content, HTTPi has a much lower computational cost compared to HTTPS for web servers.

We further discover that a significant portion of existing HTTPS content can be shared and cached across users (Section 5.3.1). This indicates that much of existing HTTPS content can be safely turned into HTTPi content to have better performance and the ability of being offloaded to other servers without any loss of security. In fact, many existing HTTPS sites contain HTTP content including scripts and images. Such mixed-content pages often contradict the intent of web sites to defend against network attackers. This is precisely due to the cost of enabling HTTPS for such existing HTTP content. It is much easier to turn HTTP content contained on HTTPS sites into HTTPi content, which will achieve the end-to-end security desired by these sites.

Although we envision a next-generation web with only HTTPi and HTTPS content, HTTP content will undoubtedly exist for a long time. We also provide web developers with an easy way to specify policies of how the three types of content can be safely mixed together (Section 5.1.2). Furthermore, we observe that the default isolation policy for HTTPi, HTTPS, and HTTP content of the same domain and port does not need to be as strict as the same-origin policy. To this end, we design a new default policy to allow useful interactions across different protocol schemes without sacrificing security (Section 5.1.3).

End-to-end authentication also requires binding a public key to an origin. Today, such bindings are established through Certificate Authorities. Recent observations have shown weakness in such CA-based binding [53]. DNSSec can potentially offer a more natural and safer way of binding a domain name to its public key [29]. We will not further discuss this topic in this thesis.

We have built an end-to-end prototype to evaluate HTTPi. On the browser side, we implemented the HTTPi protocol for Internet Explorer using IE's Asynchronous Pluggable Protocol extension mechanism. On the server side, we implemented support for HTTPi requests using an HTTP proxy sitting in front of origin web servers.

Our microbenchmark measurement indicates that HTTPi incurs an acceptable verification and one-time signing overhead, with our unoptimized implementation. This cost is quickly amortized over many requests; for example, a typical web server deployed on Amazon EC2 achieved a 4.06x higher throughput for static content served over HTTPi (and signed offline) than over HTTPS and HTTPi's throughput is negligibly lower than that of HTTP. To evaluate the efficacy of deploying HTTPi for today's web sites, we conducted an initial measurement of cacheability of today's web and found that both HTTP and HTTPS content on today's web is significantly cacheable. We also present our initial findings on the effectiveness of caching proxies to understand shared caching benefits for web users behind those proxies. Overall, our evaluation suggests that HTTPi is practical to deploy and can offer compelling benefits.

Chapter Organization. The rest of this chapter is organized as follows. Section 5.1 presents the design of the HTTPi protocol, and Section 5.2 discusses our implementation. Section 5.3 presents our measurement studies of web cacheability and an evaluation of HTTPi performance. Section 5.4 provides a brief summary of the chapter.

5.1 Design

We set the following goals for the HTTPi design:

- *Guarantee of end-to-end integrity*: Our design ensures that the integrity of the rendered content is always maintained. For example, a network attacker will not be able to inject or remove content, or have adverse impact on browser-side

rendering of content.

- *Easy adoption*: HTTPi should be easy to adopt by web sites and should fit seamlessly into the current web infrastructure. In other words, the design should be transparent to the intermediate web servers (such as CDN servers and HTTP web proxies) and should involve minimal changes to the core setup of the servers and the browser.

- *Negligible overhead over HTTP*: The design should incur negligible overhead over HTTP in computation, bandwidth, and user-experienced latency.

Note that there could be scenarios where intermediate servers modify web content, such as for personalization or content filtering in enterprises. Transmitting content over HTTPi instead of HTTP would prevent such modifications. We argue that the guarantee of integrity must be end-to-end, and any intermediate modifications should be explicitly approved by the one of the endpoints (for example, by sharing the private and public key pair of an endpoint).

To guarantee end-to-end integrity and to minimize latency and overhead, we use a content signature-based scheme that allows progressive content loading and at the same time is robust to any injection attacks, as described in Section 5.1.1. In Section 5.1.3, we describe the access control policy that browsers should carry out across HTTPS, HTTPi and HTTP content.

For easy adoption, we use the existing HTTP protocol to implement HTTPi so that intermediate web servers can cache HTTPi content seamlessly. Web browsers can show "httpi" in the address bar, but the messages on the wire speak HTTP. We use a new `Integrity` header to indicate the use of HTTPi as the protocol. The integrity header also carries the signature for HTTP headers (excluding the integrity header itself, of course). We use the existing `Strict-Transport-Security` header to prevent stripping attacks (Section 5.1.1.3) and the existing `X-Content-Security-Policy`

header to allow web sites to configure mixed content policies (Section 5.1.2). Signatures for the HTTP response body are in-band in the body itself. HTTPi's server-side and client-side implementation is pluggable into the existing setup and uses public interfaces without any need for modifying the core functionality of the server or the browser (Section 5.2).

5.1.1 Design Overview

A protocol scheme that ensures message integrity needs to satisfy two requirements. First, the identity of the server sending the content needs to be authenticated and second, the content needs to verified for integrity. HTTPi uses a content signature-based protocol scheme to satisfy these requirements.

In a strawman design, HTTPi could sign the hash of an *entire* HTTP response: The server first creates a cryptographic hash (e.g., SHA1) of the whole response and then signs the hash using the server's private key. The hash and its signature are then passed to the client along with the response. At the client side, the browser waits for the entire response to arrive, calculates its hash, and compares the value with the signed hash to authenticate the server and verify the response.

A key limitation of this design is that the browser would have to wait for the entire response to arrive before being able to verify the content integrity and dispatch the content for rendering. Consequently, this would disrupt the existing progressive content loading mechanisms in browsers, servers, and the HTTP protocol and the user would experience much longer delay before seeing any content.

We leverage previous work on content integrity [63, 65] to develop our HTTPi design that supports progressive content loading through the use of *HTTPi* segments. While these earlier efforts focused on designing protocol schemes for verification of integrity in streaming systems, our scheme is designed to be used with today's web applications and browsers. As a result, we solve new problems not addressed in

prior work, including compatibility with "chunked" transfer encoding (Section 5.1.1.1) that is widespread on the web, various content replay attacks, and stripping attacks (Section 5.1.1.3). Moreover, our work presents a detailed, practical implementation of HTTPi, whereas earlier work focused on theoretical protocol design and offered no implementation details. Before diving into our design for HTTPi, we first provide some background.

5.1.1.1 Existing Progressive Content Loading Mechanisms

Current browsers support progressive loading of web content: as soon as some data arrives from the network, the browser renders it to the user. The amount of data available at a time is determined by the underlying TCP congestion control and the network condition as well as server availability. HTTPS content can also enjoy progressive content loading especially when a stream cipher is selected by web sites.

Complementing browsers' progressive content loading, servers are also motivated to reduce user wait time and to start sending the response even before completing the processing of a request and therefore, before knowing the entire response body. To this end, servers often use HTTP chunked transfer encoding [58] and encode each piece of available response data into a chunk. A web server typically uses chunked encoding in two scenarios: (1) content is static, however, its retrieval (for example, from the server database) or processing is slow, and (2) content is dynamically generated with a chunk being a logical unit of content for the application. The chunks are sent in separate HTTP responses as soon as they are available. Note that the data of a chunk may not arrive at the client in one shot, but possibly in pieces due to network congestion. Nevertheless, the browser can consume partial chunks progressively.

5.1.1.2 HTTPi Segments for Progressive Content Loading

In HTTPi, the key challenge in supporting progressive content loading is to configure the sensible granularity of content verification. This design should meet the following

goals: (1) it leverages browser-side progressive content loading; (2) it is compatible with HTTP chunked transfer encoding; (3) it is resilient to the dynamics of the underlying TCP congestion control, which is unpredictable by servers in an offline fashion; (4) it must allow cacheability; (5) it incurs low overhead.

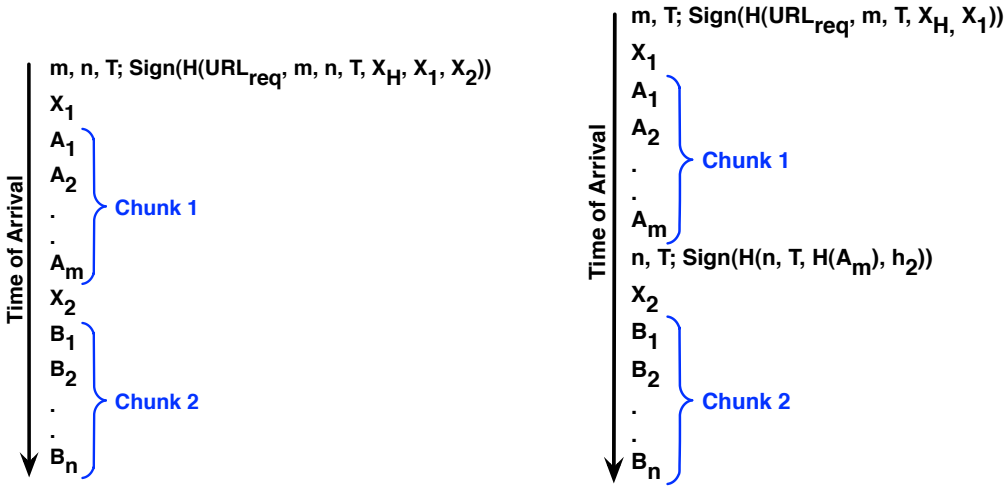
We use *HTTPi segment* to refer to the unit of verification in HTTPi. Let S denote the size of an HTTPi segment.

Using HTTP chunks as HTTPi segments would still be too coarse-grained. An HTTP chunk can be arbitrarily large and shares the same problem as the strawman solution described above.

A question one may ask is whether a server can predict how much data arrives at its clients. If so, then a server could enable verification for just that data. For a single, live connection, a server can indeed predict data arrivals on the client by obtaining the current TCP congestion control window size and the receiver window size from the network layer. However, because of dynamic network conditions, such prediction would not work well for requests at different times or from different users and would defeat cacheability. In light of this observation, S needs to be a constant value.

We choose to use the typical TCP segment size (1400 bytes) for S. TCP segment is the unit of TCP transfer. The rationale here is that the browser will need to wait for *at most* one packet to arrive to receive a full HTTPi segment, perform the verification and render the segment. This wait is as minimal as it can get.

Although HTTPi segment is the unit of verification, it does not need to be the unit of signing. In our design, we amortize the signing cost over multiple segments in the response body. In more detail, whenever a web server has some HTTP response data ready (whether it is the entire HTTP response or an HTTP chunk becoming available), for every S bytes, we take a hash, then we compute the signature for multiple hashes concatenated in the right sequence. For HTTP headers, we hash



(a) (b)

Figure 21: Protocol Scheme in HTTPi for (a) static content (b) dynamic content. $A_1, A_2, ..., A_m$ and $B_1, B_2, ..., B_n$ represent segments for Chunk 1 and 2, respectively. X_1 and X_2 represent concatenated hashes evaluated over the segments of Chunk 1 and 2, respectively. X_H represents concatenated hashes over the HTTP headers. URL_{req} is the requested URL and T is the time stamp.

each header individually and use a single signature over all hashes. Since browsers do not consume partial header values, we chose not to use the segmented design for header fields. We further amortize the signing cost by signing the hashes of HTTP headers along with the hashes of HTTP content using a single signature. We put the signature together with the sequence of the hashes at the beginning of the response body. An alternate way is to put the signature and hashes in an HTTP header. However, our scheme needs to support HTTP's chunked encoding where chunks after the first chunk do not have header fields. Therefore, we place the signature and hashes inband with the response body.

The decision on when to sign rests with the application and is made based on whether the content being signed is known in advance (i.e., static content), or is generated on the fly (i.e., dynamic content). Figure 21 gives an illustration of our protocol scheme. As can be seen in Figure 21(a), we amortize the cost of signing by using a single signature over segments for all chunks generated for static content

(e.g., X_1 and X_2 in a single signature). Since the content is known in advance, the signature and all corresponding hashes can be pre-computed by the server. For dynamic content, the hashes are computed at the time of content generation. The signature is calculated over all the segments of a single chunk (Figure 21(b)). The sequence of hashes for the headers (X_H) is placed only in the first signature. We also place the URL of the requested page (URL_{req}) in the first signature and the current time stamp (T) in each signature as a preventive measure for certain attacks (Section 5.1.1.3).

Note that signing can be done in an offline fashion for static content. For dynamic content, this incurs a computation overhead of one SHA1 computation per 1400 bytes, resulting in the bandwidth overhead of just 1.4% (20/1400). The signature overhead is one signature per chunk for dynamic content. We will show in Section 5.3 that much of the web is static and cacheable and HTTPi incurs negligible overhead over HTTP.

Any segment that fails the integrity check is not rendered. In such cases, we inform the user about the integrity failure and remove the security indicator from the page. For JavaScript, we do not perform progressive content loading because today's JavaScript engines require an entire script to be received before starting its execution.

5.1.1.3 Security Analysis and Design Enhancements

Out-of-sequence Segments. The segment hashes are arranged in a sequence before signing. If a network attacker tries to reorder the segments, it will break the sequence of the hashes and signature verification would fail.

Injection and Removal Attacks. Attacker will not be able to launch injection attacks successfully because the injected content will not be verified by the browser. Removal attacks cannot happen to the segment group of a signature for the same reason.

Nevertheless, removal attacks can happen across signature groups (a set of chunks for static content or a single chunk for dynamic content). When HTTP chunks are used by a server, each signature group will have a set of HTTPi segments and a signature for them. A network attacker can remove a signature group without being noticed at the client. To address this issue, we insert the hash of the last segment of the previous chunk at the beginning of the hash sequence of the current chunk (Figure 21(b)); and we insert the header hash at the beginning of the hash sequence of the first chunk.

Content Replay. Network attackers could also mix-and-match old content and new content to cause disruptions. Such attacks are prevented in our design by placing time stamp T in each signature. For HTTPi responses that involve multiple signatures, the browser must verify that the time stamp is the same across all signatures.

The network attackers could alternatively replay a completely different response for requested object. In order to correctly identify the response with the requested object, the client verifies its own value of the requested URL against the signed URL_{req} value.

Stripping Attacks. Both HTTPS and HTTPi are prone to "stripping" attacks that hijack a user's initial insecure HTTP request and remove redirects to secure content. Although it is possible to notice stripping attacks by manually checking the browser security indicators, users often ignore these indicators [117]. The HTTP Strict Transport Security protocol (HSTS) prevents these attacks by allowing web sites to specify a minimum level of security expected for connections to a given server. The policy can be delivered via HTTP header [71]. To prevent attacks on the user's very first visit to the site, the policy can also be delivered via DNSSEC [79]. We use an extension to HSTS, `allowHTTPi`, to allow severs to specify HTTPi as the minimum level of security. The `allowHTTPi` token is appended to the server's existing `Strict-Transport-Security` policy declaration. Older browsers that do not support

HSTS will ignore this header, while older browsers that support HSTS but not our extension will default to HTTPS for all content.

Denial of Service. HTTPi is limited in its capability to handle denial of service attacks, where a network attacker strips off the integrity header from the response that requires integrity as specified by the application (Section 5.1.2). As a result, the content would not be rendered by the browser. Additionally, the attacker can allow some segments to be rendered, while preventing subsequent segments to arrive through to the browser. This could potentially corrupt the internal logic of the application. For example, the attacker can strip off JavaScript that changes the content of the page and as a result, the page remains rendered in its original form. One possible countermeasure to this attack is to use a time out for inter-segment arrival at the client and raise an integrity failure alert after the expiration of the timer. However, it would require an estimation of the typical inter-arrival time for each client, which might not always be accurate. In our design, we allow the browser to wait infinitely for the packets to arrive. If the user clicks on stop, we alert the user that the content is not complete. Since we do not execute JavaScript till it is fully received, partially rendered JavaScript would not be an issue for the integrity of the site.

5.1.2 Mixed Content

The mixed content condition occurs when a web developer references an insecure (HTTP) resource within a secure (HTTPS) page. Such references create vulnerabilities that put the privacy and integrity of the otherwise secure page at risk, because the insecure content could be modified in network transit. Scripts are particularly problematic because they acquire the principal origin of the including page, allowing malicious scripts to read or alter the content that was delivered over the secure connection. These types of vulnerabilities are becoming increasingly dangerous as

more users browse untrusted networks and attackers improve upon DNS poisoning techniques and weaponize exploits against insecure traffic.

Browsers differ in their mixed content handling. Internet Explorer prompts the user before displaying mixed content, while Firefox and Google Chrome show a modified browser lock icon. From a security standpoint, the best behavior would be to block all insecure content in secure pages without prompting the user. The latest beta release of IE9 enforces this behavior on scripts and stylesheets, but not images; this policy is similar to the one proposed by Gazelle [129]. However, this option of automatically blocking insecure content has some serious compatibility implications. It might potentially confuse the user, since pages that rely on insecure resources could appear broken. In the worst case, the user might think the broken pages indicate a bug in the browser and subsequently switch to an older version of the browser or to a completely different browser to get unbroken pages.

We argue that mixed content vulnerabilities should be fixed by the web developers, both for security and user-experience reasons. The web developers have a better understanding of the impact that embedded content can have on the security of their site. Additionally, they are in much better position to develop a user-friendly fallback mechanism for their site in case some content fails security checks and hence is not rendered.

By default, we require that all active content embedded in HTTPi and HTTPS pages, such as scripts and stylesheets, be rendered over HTTPi or HTTPS. To allow web applications to customize this default behavior, we use an HTTP header that is compatible with Content Security Policy (CSP) [125] header to specify the server's end-to-end integrity requirements for dependent resources. The CSP policy syntax is convenient for our purposes as it already allows sites to specify which origins they want to include content from.

An example policy is as follows:

```
X-Content-Security-Policy:
     allow https://login.live.com
           httpi://*.live.com:443
```

The above example informs the browser that all embedded objects from `login.live.com` should be retrieved over HTTPS and content from all other subdomains of `live.com` needs to be downloaded over HTTPi. If the servers hosting the embedded content do not support the corresponding protocol, then the content is considered unsafe as per the web page's requirements and hence should not be rendered by the browser. Our design also supports specification of integrity requirements at a finer level, i.e., at the level of object types or specific objects themselves. However, the web application should be careful in specifying such finer policies as it increases bookkeeping at the server. It also has the potential to break existing interactions within the embedded content if the policies are not correctly specified.

The CSP syntax provides an ideal mechanism for the web developers to handle mixed content. It does not require web applications to change their code by explicitly modifying all insecure references of embedded objects. Even if web developers decide to modify their code, it might not be sufficient. A secure (HTTPS or HTTPi) URL can still return a redirect to an insecure resource, which could be difficult to determine by examining the DOM alone. Additionally, a script delivered over a secure channel could still make references to insecure content. In our design for HTTPi, the browser enforces the policies specified by CSP for all statically or dynamically generated URLs.

5.1.3 Access control across HTTPS, HTTPi, and HTTP content

HTTPi content can be embedded in an iframe through the use of the "httpi" scheme, such as <iframe src="httpi://a.com/">, or through the use of an additional iframe "integrity" attribute, such as <iframe src="http://a.com/" integrity>. The former has the consistent presentation with other protocol schemes. The latter has the

benefit of backward compatibility; on an older browser, HTTPi content would simply render as HTTP content. Note that no matter what the representation is, the network messages still speak HTTP to be backward compatible with the existing web caching infrastructure.

The Same Origin Policy labels the principals with the origin defined as the triple of <protocol, domain, port> [128,129]. Therefore, content from the same domain and port number but with different protocol schemes is rendered as separate principals. They can only communicate explicitly through messages (i.e., `postMessage`) [32]).

In this subsection, we consider the default interaction and access control model for HTTPS, HTTPi, and HTTP content served from the *same* domain and port. For example, a top-level HTTPi page may embed two iframes, one containing HTTP content and the other containing HTTPS content; and all three pages are from the same domain and port. While following the SOP is safe for such scenarios, it disallows all interaction among HTTP, HTTPi, and HTTPS content. Rather than accessing the DOM objects directly, developers would be forced to redesign such interaction with asynchronous `postMessage`-based protocols, which may be hard to design correctly, as illustrated by recent flaws found in Facebook Connect and Google Friend Connect [68]. As a result, a developer may be discouraged from converting some content on an HTTPS site into HTTPi to benefit from its cacheability properties.

As a concrete example, consider an online shopping site that is rendered over HTTPS to protect users' private data such as credit card information. The site presents users with a map to select a site-to-store pick-up location during checkout. It may be desirable to deliver the store information and map content over HTTPi, but this raises a problem of allowing the HTTPS part of the site to read the store selection made by the user, an interaction that would be disallowed by SOP. As a result, the site's developers may be forced to refactor their code to use `postMessage`.

We observe that the SOP semantics are more restrictive than actually required

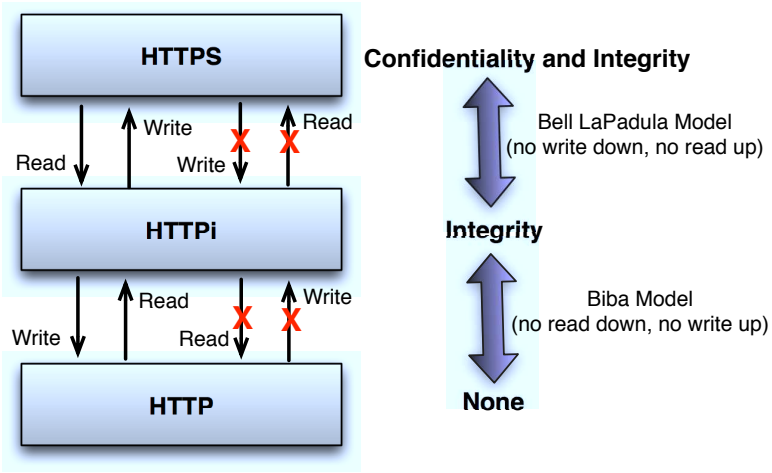


Figure 22: Interactions in Mixed Content Rendering

to ensure security for such scenarios. Our goal is to allow legitimate communication while preserving the security semantics, namely the confidentiality and/or integrity, of the rendered data. Our default communication policies are inspired by the combination of the Bell LaPadula [33, 35] and Biba [39] models. It is important to note that our goal is *not* to enforce information flow invariants often associated with those models (e.g., frames of any origin can already freely communicate via `postMessage`), but rather to use these models to determine a secure and convenient *default* isolation policy for our setting. We summarize these models as the following set of rules:

Bell LaPadula model (for confidentiality):

- The Simple Security Property: a subject at a given security level may not read an object at a higher security level (no read-up).
- The *(star) property: a subject at a given security level must not write to any object at a lower security level (no write-down).

Biba model (for integrity):

- The Simple Integrity Axiom states that a subject at a given level of integrity may not read an object at a lower integrity level (no read down).

- The * (star) Integrity Axiom states that a subject at a given level of integrity must not write to any object at a higher level of integrity (no write up).

In view of these models, we represent the three protocols (HTTP, HTTPS and HTTPi) by two confidentiality levels (C_{high} and C_{low}) and two integrity levels (I_{high} and I_{low}), which models the high and low requirements for confidentiality and integrity, respectively. HTTPS can be realized by the tuple $<C_{high}, I_{high}>$, HTTPi by $<C_{low}, I_{high}>$ and HTTP by $<C_{low}, I_{low}>$. Using this model, we define the access control rules across HTTP, HTTPi, and HTTPS as follows:

HTTPS and HTTP. HTTPS' confidentiality label C_{high} is higher than HTTP's confidentiality level C_{low}, thus resulting in "no read up, no write down" requirement of the Bell LaPadula model. The integrity levels of HTTPS and HTTP, I_{high} and I_{low} respectively, with $I_{high} > I_{low}$, results in "no write up, no read down" condition of the Biba model. Combining these two requirements results in no reads or writes to either side being allowed between HTTPS and HTTP. This derivation is consistent with the SOP.

HTTPi and HTTP. Since confidentiality levels of HTTPi and HTTP are equal, only the integrity levels enforce the "no write up, no read down" policy from the HTTPi content to HTTP resources (Figure 22). Firstly, this means that a script belonging to the HTTPi principal can write to the HTTP part of the page without reading its content. One reason to prevent an HTTPi script from reading HTTP content is to prevent the HTTP input from influencing the logic within the HTTPi content. However, an HTTPi script might still desire to read the HTTP page to identify the DOM element to write to. So, our requirement is to allow the read operation on the HTTP content without allowing the logic of HTTPi content from being affected. One way to realize this is by performing complete information flow check in the HTTPi code, which might not be practical. We use an alternative approach in which the HTTPi content itself writes the code for reading the HTTP

content, and this code is injected into the HTTP content. This injected code runs within the HTTP principal and hence can freely read and write to the content. Since HTTPi relinquishes the transferred code to the HTTP integrity level (I_{low}), that code cannot affect the logic of HTTPi's own code, though it still can read from HTTPi content. Secondly, HTTP can read the HTTPi content, but cannot write to it. We realize this in our design by providing only a shadow copy of the HTTPi content to HTTP, with no direct reference to real HTTPi objects.

HTTPS and HTTPi. Since HTTPS and HTTPi integrity levels are equal, only the confidentiality levels force the "no read up, no write down" rule from HTTPS to HTTPi resources (Figure 22). Both read and write operations can be realized similarly to the previous scenario. We allow HTTPi content to write to HTTPS since the code for HTTPi is at the same integrity level as HTTPS content and written by the same developer (since they have the same domain). HTTPi scripts can write the code for reading the HTTPS content into the HTTPS' DOM and effectively, that code becomes part of the HTTPS principal. This allows reading of the HTTPS code by the injected code without leaking any of the read data back to HTTPi's main code. For reading HTTPi content without allowing any write, a shadow of the HTTPi's DOM is provided to the HTTPS. Coming back to the shopping site example earlier in this section, this rule would allow HTTPS content to read the store selection made by the user and correspondingly send the merchandise to the selected store.

5.2 Implementation

HTTPi requires both the client browser and the hosting server to adhere to the protocol. Accordingly, our implementation consists of server-side and client-side modules. Figure 23 shows the high-level architecture of our system. Our server-side implementation consists of an HTTPi Transformer, which implements all HTTPi-related

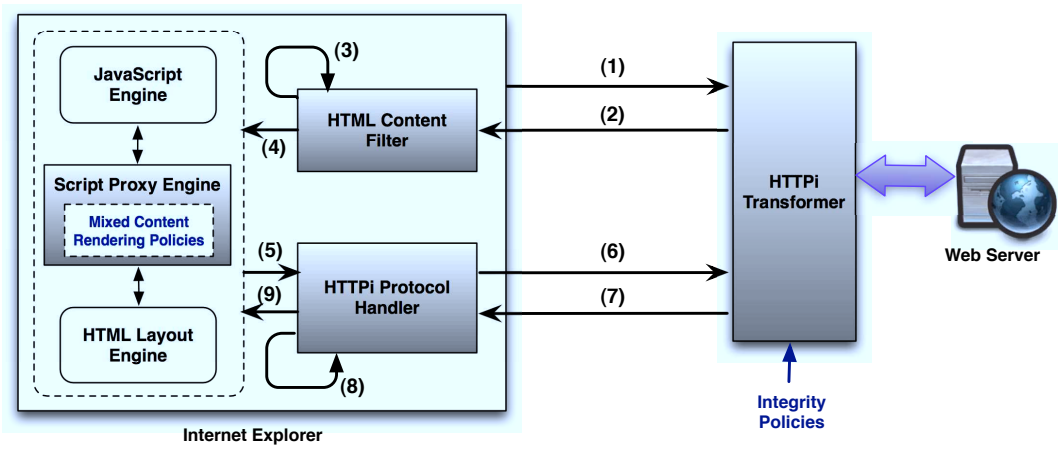


Figure 23: High-Level Architecture of our HTTPi Implementation with the operational steps to retrieve content over HTTPi as follows: (1) IE makes an initial request for a specific page. (2) Server-side proxy identifies that the request is for a HTTPi-enabled resource and appends integrity policy headers to the response. (3) HTML content filter processes the response by modifying URLs that match STS policies to point to their corresponding HTTPi links. (4) HTML content filter releases the modified response to IE's rendering engine. (5) The HTTPi protocol handler is invoked for every HTTPi object encountered during rendering. (6) The HTTPi protocol handler makes a HTTP call to the server requesting the object. (7) Server-side proxy traps the request, makes an independent HTTP call to the backend web server to get the response, hashes and signs the response, and returns it back to the HTTPi protocol handler. (8) The HTTPi protocol handler verifies the signature and hashes corresponding to the different segments in the response. (9) Successfully verified segments are passed to the rendering engine for progressive loading. The Script Engine Proxy (SEP) subsequently mediates all mixed-content interactions while a web page renders.

interactions on the server side, including content hashing, segmentation, and a handler for appending integrity policy requirements to HTTP responses. Our client-side implementation centers around three modules that we add to Internet Explorer 8: (1) an HTML content filter that transforms a given page to adhere to integrity policy requirements, (2) an HTTPi protocol that handles the client-side processing of HTTPi content, and (3) a module that provides JavaScript and DOM interposition to enforce our mixed-content access control policies. In this section, we describe each of these modules and the associated implementation challenges in turn. Overall, our implementation consists of 1,100 lines of server-side code, and 3,500 lines of client-side

code.

5.2.1 Server-side Implementation

We explored two options for implementing the server-side component of HTTPi, with the options differing in their deployment tradeoffs. First, we extended the IIS 7 web server with a C# module for HTTPi, called HTTPi Transformer, that encapsulates the functionality to generate HTTP responses with signatures and content hashes that adhere to the HTTPi protocol. Although we chose IIS, similar module functionality is available for other web servers. This option is useful if the server is willing to immediately integrate HTTPi functionality into their current setup. It also has obvious performance benefits as the module is closely coupled with the functionality of the web server.

In our second deployment option, we integrated the HTTPi Transformer into a web proxy that translates typical HTTP responses into HTTPi responses by embedding all the hashes and signatures needed by HTTPi. We leveraged the public-domain Fiddler web debugging proxy [84] and its FiddlerCore [57] extensibility interfaces. This option is independent of web server implementation and allows servers to continue supporting HTTP as the delivery protocol for backward compatibility, while switching to the HTTPi protocol for requests that pass through the proxy. It eases deployment, since the proxy can be deployed anywhere in the network and guarantees integrity between the proxy and a compatible browser. This could be desirable for corporations that do not require integrity checks for intranet users, while ensuring integrity of their sites for external users.

For our evaluation, we used the latter option of having a network proxy, because (1) it allowed us to test our prototype against publicly deployed web sites without having any control of their web servers, and (2) it allowed fair comparison of HTTPi with HTTPS and HTTP (Section 5.3.2.3) by cleanly switching to a desired protocol

between the client and the proxy even when the backend server did not support the protocol.

5.2.2 Client-side Implementation

5.2.2.1 Filtering content to enable HTTPi

We expect that origin servers would generate new content with the right "httpi" URIs for the content that requires integrity. In any case, our design ensures that mixed content policies are enforced by verifying the URIs against the policies. Instead of requiring the servers to change the URIs in their existing content, our implementation of HTTPi performs the required filtering to enforce the mixed content policies. The HTML content filter module is invoked for every HTML response received at the browser that is associated with a Strict Transport Security policy or Content Security policy. This module modifies HTML content to ensure that it adheres to the minimum security levels specified in STS and CSP. For example, all object links on a page are transformed to corresponding HTTPi links by modifying the protocol field in the URL. Since the HTML content filter is invoked before the page is rendered in the browser, this design allows the HTTPi protocol handler to be associated with all such links and hence ensures that the HTTPi handler is invoked when the browser requests those links during rendering. We implemented this module by using IE's public MIME filter COM interfaces [1] and subsequently registered it as a filter for HTML content.

One limitation of this approach is that it may miss dynamically-generated links where the URL is constructed by JavaScript at runtime. We are currently working on solving this by performing HTTPi redirection to the time of actual HTTP requests; our evaluation is independent of this implementation enhancement and was performed without it.

5.2.2.2 HTTPi Protocol

The HTTPi protocol handler encapsulates all client-side handling of HTTPi content and is automatically invoked by the browser when an HTTPi link is encountered by the browser's rendering engine. Upon invocation, it makes an independent HTTP call to the server to retrieve the content. It then verifies the integrity of the content in segments using the mechanism described in Section 5.1. Once the integrity of a particular segment is verified, its content is released to the browser's rendering engine for progressive loading.

We implemented this module as an asynchronous pluggable protocol (APP) [1] IE module associated with the HTTPi protocol. Even though IE provides this generic protocol extension point, implementing a general-purpose protocol with minimal performance overhead is challenging. IE's internal logic is well-optimized for HTTP and HTTPS, which makes a comparably performant web protocol difficult to implement. A considerable time and effort was spent on making our code as optimal as possible by parallelizing various operations such as network read and signature verification. Despite our limited knowledge of IE's internal optimizations and with the handicap of using a generic interface, we were still able to achieve acceptable performance as compared to HTTPS and HTTP (Section 5.3.2.3).

5.2.2.3 Access control for mixed content

Another big challenge for our implementation was to customize SOP to include our mixed-content access control policies. Unfortunately, IE does not allow changing the code for SOP with public APIs. As a result, the only alternative was to implement our solution as an additional layer on top of the existing SOP and then find a way to enforce mixed-content policies within the limits imposed by the existing SOP logic. This certainly made our implementation more difficult.

To solve this problem, we use a two step approach. In the first step, we modify

the security origin (origin is defined as the tuple `<protocol, domain, port>`) of all objects on the web page by changing the protocol field to HTTP, i.e., the one with the lowest integrity and confidentiality level. This is achieved by providing a custom implementation for the `IInternetProtocolInfo` interface [12] from within the APP for HTTPi. Note that changing the security origin of an element does not affect the URL associated with that element.

As per the SOP, all the objects on the page can now interact without restriction. Our second step is to enforce access control rules or policies that govern such interactions. We build on our earlier work [123, 128] that implements a JavaScript engine proxy (called script engine proxy or SEP): SEP is installed between IE's rendering and script engines, and it mediates and customizes DOM object interactions. SEP is implemented as a COM object and is installed into IE by modifying IE's JavaScript engine ID in the Windows registry. We extend SEP to trap into all invocations (read or write) across the page's objects and ensure that our mixed-content access control polices (Section 5.1.3) are enforced. We use the URLs associated with the accessing object and the object being accessed in making our access control decision. The two-step logic that governs the access control enforcement in our implementation can be summarized as follows:

- If the original origins of the caller and the callee objects differ in `domain` and/or `port`, then the browser would prevent any interactions across them in accordance with the SOP.

- If the original origins of the caller and the callee objects differs in only `protocol`, the SOP would allow the objects to interact (as we modify the protocol of the security origin to HTTP). In this case, we mediate the interaction within our customized SEP to enforce our access control policies.

The read operation is straightforward: SEP allows the caller to have read access to

Table 13: Measurement of publicly cacheable web content from the top 1000 Alexa sites.

Protocol	Total Objects		Publicly Cacheable Objects	
	Count	Size	Count	Size
HTTP	346,629	1532 MB	251,826 (72.65%)	1385 MB (90.41%)
HTTPS	5,036	21.95 MB	3,659 (72.66%)	19.39 MB (88.33%)

the callee's objects. The write operation could be implemented in a similar fashion; however, some writes must first access an object to which the write subsequently occurs. For example, if the caller wants to write content to a specific element on a callee object, it might need to read the handle to that element using functions such as `getElementById` or `getElementsByName`. However, if the caller only has write privileges with no read access, it cannot make such calls and hence cannot know where to write the content.

We solve this problem by introducing a new JavaScript function `writeUsingCode`, which is interpreted by our SEP implementation; the browser's JavaScript engine does not need to understand this function. Instead of directly making read calls looking for an element of the callee object, the caller uses the function to pass the JavaScript code that encapsulates such read calls and the subsequent write call to the corresponding element. The SEP intercepts this function call and makes calls to the underlying JavaScript engine to execute the code with the origin of the callee object. Any unintended feedback mechanism introduced by this code is prevented by SEP's access control policies.

5.3 Evaluation

We have implemented a HTTPi system that works end-to-end. We used our proxy-based implementation as a server-side HTTPi endpoint to verify our system for correctness against a number of popular web sites, such as Google, Bing Maps, and Wikipedia. In each case, the browser successfully rendered the web pages and all

integrity checks were correctly included at the server and verified at the browser. Any tampering of the web page in the network was correctly detected and failed the integrity check at the browser. We evaluated the access control interactions for mixed content by developing a set of custom web pages that included such interactions. Our system correctly enforced the access control policies for such interactions.

Next, we provide experimental evidence to support our claim that today's web sites can benefit from cacheability enabled by HTTPi. To this end, we first perform a web cacheability study to answer two questions: (1) what web sites have cacheable content, and (2) what users are taking advantage of shared caches on the web. Next, we evaluate the performance of our prototype by micro-benchmarking its operations and by comparing its overhead to that of HTTPS and HTTP.

5.3.1 Study of Web Cacheability

With HTTPi, web sites decide what content uses HTTPi as the underlying mechanism of transport. Therefore, any content that web sites currently allow to be cached by intermediate web servers, such as CDNs and web caches, becomes an ideal target for HTTPi. To better estimate the amount of web content that could benefit from the use of HTTPi, we performed a cacheability analysis on the top 1,000 Alexa sites that includes both top-level pages and embedded content on the sites visited. We analyze the HTTP caching headers, such as `Cache-control`, `Expires`, `Pragma`, etc., to decide what content is deemed cacheable according to the HTTP specification [58].

Experimental Setup. To facilitate automatic analysis for a large number of URLs, we used a customized crawler from our earlier work [123], which utilizes IE's extensibility interfaces to completely automate the browser's navigation. To invoke functionality beyond a site's home page, the crawler uses simple heuristics that simulate some user interaction, such as clicking of links and searching form submissions. We restrict all simulated navigations to stay within the same origin as a site's

home page. We monitor the browser's network traffic in a proxy to intercept all HTTP/HTTPS requests and analyze HTTP headers relevant to web caching. The proxy is included as a trusted certificate authority at the browser in order to allow it to intercept the HTTPS traffic and inspect its content [84].

Prevalence of cacheable content. Table 13 shows the results of our web cacheability experiment. Note that our results only consider content that is marked as public and excludes any private content that is user-specific and hence is intended to be cached only at the user's browser. As we can observe from the table, a large majority of the web content is rendered over HTTP with more than 98% of the objects that we observed being HTTP objects. We found that approximately 73% of these objects are cacheable. The cacheability is higher in terms of content size, with more than 90% of total HTTP content size (of all objects) being cacheable, indicating that the web applications typically want larger-sized content, such as images, to be cached in the network. The limited number of HTTPS objects that we encountered follow a similar trend with a large number (73%) being cacheable objects. The presence of a considerable number of public, cacheable HTTPS objects is an indication that web applications intend to cache objects in the web, but are discouraged by the lack of security provided by HTTP. They are left with no choice but to trust the CDNs for this type of content. If only integrity of the content is desired, HTTPi presents itself as an ideal alternative for these HTTPS objects.

Presence of in-network caches. To see how many users are benefiting from web caches today, we measured the prevalence of forward caching proxy servers, which are a significant source of in-network caching. More specifically, we conducted an experiment to determine how the country and the user agent affects whether a forward network proxy is being used. We used rich media web ads as a delivery mechanism for our measurement code, using the same ad network and technique previously demonstrated in [77]. We spent $80 to purchase 115,031 impressions spread across

194 countries. Our advertisement detected forward proxies using XMLHttpRequest to bypass the browser cache and store content in the network cache. Overall, 3% of web users who viewed our ad were using a caching network proxy. However, some countries had a significantly higher fraction of users behind network proxies. Popular countries for forward proxies included Kuwait (63% of 372 impressions), United Arab Emirates (61% of 624 impressions), Argentina (11% of 1,875 impressions), and Saudi Arabia (10% of 4,248 impressions). We also observed higher usage of forward proxy caches (11%) among mobile users, although these users accounted for only 0.1% of the total impressions in our experiment.

Relevance to HTTPi. Our results demonstrate that cache proxies are still prevalent and useful today, particularly for large user communities, such as a whole country of people behind a single firewall and mobile users behind cellular gateways. HTTPi can take advantage of these proxies while offering end-to-end security at the same time.

5.3.2 Performance Evaluation of HTTPi

We evaluate the performance of HTTPi in two steps. First, we perform micro-benchmarking of various stages of the protocol and analyze the parameters that determine HTTPi's performance. Second, we analyze the end-to-end performance overhead of HTTPi over existing HTTP and HTTPS protocols.

5.3.2.1 Experimental Overview

Ideally, we would run performance experiments on real web sites deployed on the web. However, current web servers do not understand the HTTPi protocol, and many servers host an HTTP version of a site but not HTTPS. To overcome this, we used our modified server-side Fiddler [84] proxy (Section 5.2.1) for proxying all requests from the client to the backend server, and converting HTTP requests from the origin server into HTTPi or HTTPS requests to the client, as necessary for our

experiments. This setup allows us to measure the cost of using HTTPS and HTTPi for web pages that are currently hosted over HTTP.

We use the end-to-end response time as the measurement criterion, defined as time between the instance at which a URL is submitted at the browser and the instance at which the corresponding page is fully rendered. To remove any discrepancies that might arise from fetching content from the backend server due to inconsistent network conditions, we deduct the data fetching time at Fiddler from the total end-to-end response time. This gives us an estimate of the end-to-end response time with Fiddler acting as the server. For a fair comparison, we also perform similar deductions for HTTP and HTTPS.

For our experiments, we use SSL certificate size of 1024 bits. Even though there is a push on the Internet to move towards 2048-bit certificates, many of the popular sites such as Gmail still use 1024-bit keys. Additionally, it makes HTTPi's performance estimates to be conservative in comparison to HTTPS, as HTTPS will perform worse for 2048-bit keys.

Using the Akma network delay simulator v0.9.129 [17], we simulated various network conditions to understand their performance impact on end-to-end response time. We simulate the incoming and outgoing connections to have equal bandwidth and fixed their queue sizes at 20 packets. We run our delay simulator on the server side to cap the server throughput to a desired bandwidth. We deploy our server-side Fiddler code on a Windows 7 machine, with an Intel 2.67 GHz Core i7 CPU and 6 GB of RAM. The client runs on a Windows 7 machine, with an Intel 2.4GHz quad-core CPU and 4GB of RAM. All experimental results are averaged over 10 trial runs.

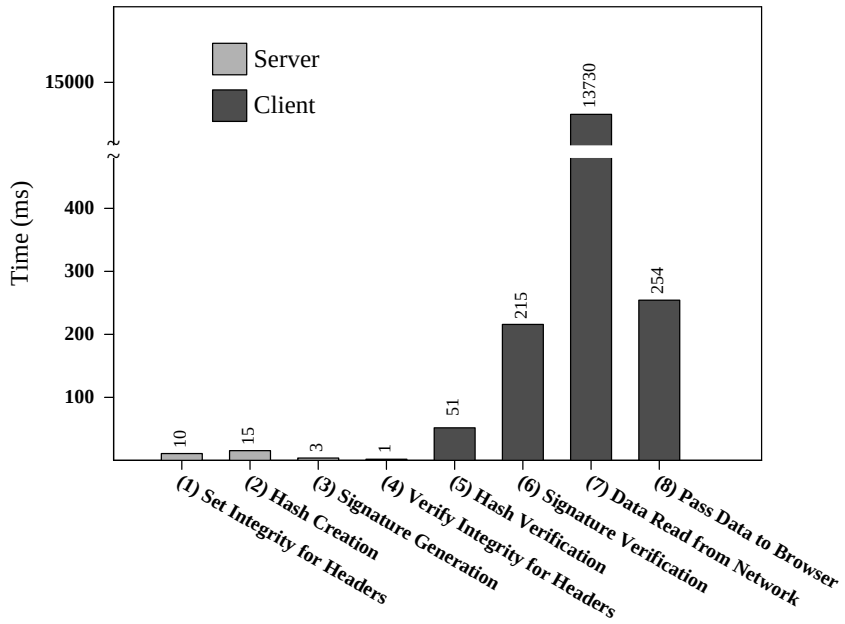


Figure 24: Micro-benchmarking various operations in HTTPi for a 836KB web page, using 512Kbps network bandwidth.

5.3.2.2 Micro-benchmarks

To understand sources of overheads in our system, we instrumented our HTTPi implementation to measure latencies of various operations, and used a simulated network bandwidth of 512Kbps to load an 836KB HTML page in our HTTPi-enabled browser, with the size picked to maximize measurable overhead and to observe effects of HTTPi's segmentation. Figure 24 breaks down the delays contributing to the end-to-end response time, which we measured to be 15.7 sec.

We find that a large fraction of the total time is spent reading content from the network (bar 7 in Figure 24), which is an expected behavior for slower networks. The overhead costs of hashing all content segments (bar 2) and signing these hashes with a 1024-bit key (bar 3) on the server side is very small. Here, the RSA signature is calculated on a fixed-size single SHA1 hash of 20 bytes (Section 5.1); this takes just 3ms. Since the header value sizes are much smaller as compared to the content body, both the time to set the header integrity content (hashing and signing) on the

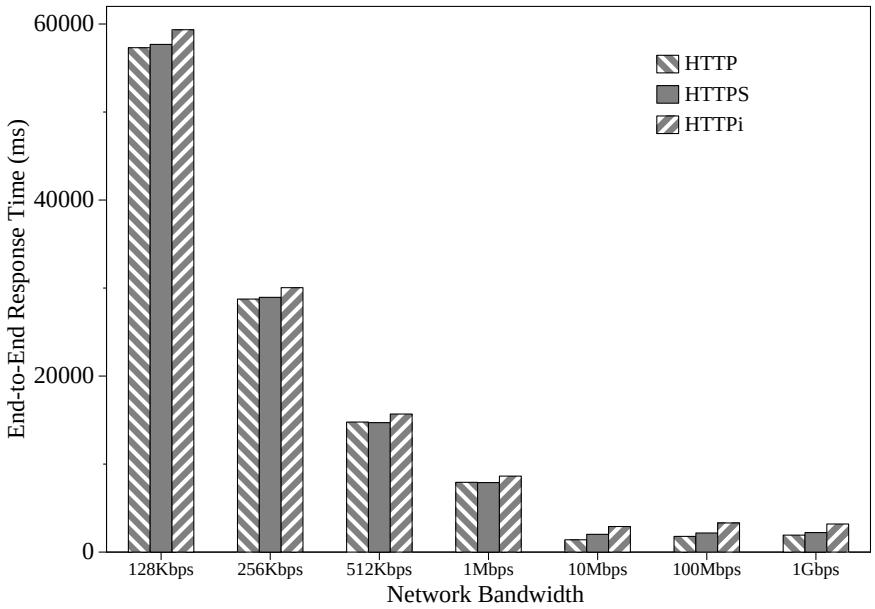


Figure 25: End-to-end response time as a function of the network bandwidth available to the client, measured for a 836KB page. Note that these results do not include performance benefits due to caching for HTTP and HTTPi.

server (bar 1) and time to verify it on the client side (bar 4) is low.[2] On the client side, the signature verification time (215 ms, bar 6) is a more significant source of overhead. It is considerably higher than the cumulative hash verification time for all content segments (51 ms, bar 5), supporting our design of using a single signature over multiple segment hashes. The time to pass data from our client-side HTTPi protocol handler into the browser's rendering engine (bar 8) is also considerable; although it is not specific to HTTPi and would also be incurred by other protocol handlers in the browser, native protocols like HTTP are more optimized in our browser for this step, as we discussed in Section 5.2.2.2.

In summary, we find that the major HTTPi components (bars 1-6) constitute only 295 ms (1.8%) of the end-to-end response time for this microbenchmark, with largest overhead coming from client-side signature verification.

[2]Note that we do not perform any segmentation for headers. For our measurements, we specify two headers, `Server` and `Content-Type`, to require integrity. This time cost will vary according to the number of headers for which integrity checksum is set.

5.3.2.3 Comparing HTTPi to HTTP and HTTPS

In this section, we compare HTTPi's performance to that of HTTP and HTTPS and answer two questions: (1) Is the user-perceived latency acceptable for the data received over HTTPi, and (2) What is the performance impact of running HTTPi and the hashing and signing load it incurs on a web server?

User-perceived latency. We compared the end-to-end response time for our 836KB test page rendered over HTTPi, HTTPS and HTTP. Figure 25 shows the results of our experiments performed over different network bandwidth conditions. Note that the performance results do not include caching, and only evaluates the first of potentially many requests for this page. Evaluating performance of a particular cache is not a goal of our experiments and has been previously well studied [130, 133]. We see that HTTPi incurs minimal overhead over both HTTP and HTTPS, and this overhead is consistently within 0.7-2.0 seconds over both HTTP and HTTPS for different network bandwidths. Since this value does not vary much with network bandwidth, we believe our implementation is successful in approximately matching the network optimizations of HTTPS and HTTP. We believe that there is still ample room for client-side optimization as we discussed earlier in Section 5.2, and this will certainly reduce the total overhead of HTTPi (since client-side overhead is not negligible as shown by our micro-benchmarking experiments).

Web Server Throughput. Our server throughput measurements are performed using httperf [94], an HTTP performance measurement tool. The experiments are performed using two different setups that closely represent typical real-world web deployments:

- Our first setup consists of an IIS server that is hosted on a bare-metal Windows 7 machine, with Intel 2.67 GHz Core i7 CPU and 6 GB of RAM. The Linux client machine running httperf is connected to the server by a 1Gbps network with negligible latency.

Table 14: Impact of HTTPi and HTTPS on server throughput in responses/sec.

Experiment	HTTP	HTTPi	HTTPS
Bare-metal Setup	3320	3318	2503
Amazon EC2 Setup	2757	2732	678

- Our second setup is cloud-based; we use a virtual Windows 2008 Server image on Amazon EC2. At the time, this image was the only publicly available image that came pre-installed with IIS 7. It is a "high-CPU medium" instance with 5 EC2 compute units with 1.7 GB of RAM (the fastest instance that was available for this image). This setup mimics a typical EC2 user who wants to host a web server. httperf is executed from a Linux EC2 instance in the same region, using EC2-private LAN with negligible network latency.

We use an experimental HTML page of size 4.8 KB, which represents a typical size of a page with no embedded links. We arrived at this page size based on the web estimates that put total page size at 170KB (median) and number of objects per page at 37 (median) [107]. For each page, we increased the offered load on the server until the number of sustained sessions peaked. We found that the server was CPU-bound in all cases. Each session simulated one request to the web page.

Table 14 shows a summary of our results. HTTPi incurs negligible degradation (less than 1%) of throughput compared to the original HTTP page. In comparison, the throughput drop was substantial when using HTTPS, with our bare-metal experiment reporting 25% and EC2 experiment showing 75% drop in the throughput. This drop is attributed to the heavy CPU load for the SSL handshake. Our bare-metal experiment shows a lesser drop since it has a considerably faster CPU, which handles the load better. Overall, these results demonstrate that web servers can have a significant performance incentive to use HTTPi instead of HTTPS.

5.4 Summary

We envision HTTPi to complement HTTPS to bring end-to-end security to the entire web. Only when there is end-to-end security, the browser platform and the web are able to have a collectively secure overall system.

We advocate the part of web that does not have end-to-end security today to adopt HTTPi which incurs negligible performance overhead over HTTP and enjoys the benefit of CDNs and cache proxies just as HTTP. For existing HTTPS content, our study indicates that its significant portion is cacheable and can also gain significant performance and caching benefit from employing HTTPi.

CHAPTER VI

RELATED WORK

6.1 Access Control

Access control is defined as any mechanism by which a system grants or revokes the right to access some data, or perform some action. It is a well established area of research and previous work in this field range from creation of new models [42,51,115] to improving improving various aspects of these models [86–88, 141]. In this thesis, our focus is not on creating yet another all-purpose access control model but is on developing a generalized framework to allow such models to be integrated into web applications (Chapter 2).

Access control for web applications and services is an area that has been well studied in the literature [43,49,90]. While such centralized solutions are well suited to the web applications, their design is limited in the changing Web 2.0 paradigm towards user-contributed data. Our work is more user centric allowing users to contribute in defining the access control for their own data.

The need for user-defined policies for user applications is being understood [64, 120]. PinUP is one system that allows users to manage file access of his own applications in an isolated environment specific to the user [54]. In another work, Simpson has argued that the users of social networking sites should have the opportunity to construct fine-grained access control policies that meet their particular requirements and circumstances [120]. Chinaei et. al proposed a decentralized access control system in which corporate policy can allow all health record owners to administer access control over their own objects [44]. Our work is not specific to any type of application and can be integrated into a wide range of web applications, including social

networks, content sharing sites, and many others.

Gates has proposed use of access control based on the real-world relationships of users [64]. In agreement with our work, she also favors access control decisions to be made by the users regarding access to their data. She further argues the access control policies and relationship groups defined by the user should follow the user, rather than be redeveloped for each individual site. While her ideas relates to some of the work described in this thesis, there is no real implementation of the ideas presented in her work.

There are some proposed solutions that allow inter operability between diverse web applications by using user-centric identities for access control mechanism. Lockr is an access control system based on social relationships that lets people manage their social networks by themselves in one place (e.g., through their personal address books) while letting web sites and Internet systems be in charge of content delivery only [127]. This eliminates the need for users to maintain many site-specific copies of their social networks. Similar protection is achieved by the single sign-on decentralized model of OpenID [18].

6.2 Information Flow Control

Information flow control at the language level has been well studied [45, 98]. Jif is a Java-based programming language that enforces decentralized information flow control within a program, providing finer grained control than xBook [98]. In comparison to these language level techniques that require the applications to be rewritten, the xBook platform provides a simpler interface to the application programmers: they do not need to learn a new language or perform any fine-grained code annotations. Additionally, information flow on a language like JavaScript with dynamically created source code may not be feasible. Cong et al. [45] presented a technique of writing secure web applications, which generates JavaScript code on the client side and java

code on the server side. However, the applications are still written in the Jif language.

There are other systems [82,140] that have utilized the information flow concept to control data flow at the operating systems (OS) level. Information flows are tracked at low-level OS object types such as threads, processes, etc. xBook works at a much coarser level at the applications, with smallest unit of information being an application component. As a result, run-time information flow in xBook would probably be less expensive as compared to a much finer granularity level used in these systems. In order to make these systems useful for a typical social networking environment, it would require the systems to be installed at a user's computer because leaks can also happen at the browser, which might not be feasible. In comparison, xBook runs on a typical web server without any changes to the OS environment.

Similar to the ADsafe environment, other safe subsets of programming languages, such as JoeE [59] (for java) and Caja [91] (for JavaScript), allow third-party applications to provide active content safely and flexibility within the existing web standards. While we used ADsafe for its simplicity and suitability to meet our system needs, we expect that it would be similarly possible to develop xBook using these alternatives.

6.3 Browser Access Control Policies

We are not the first to find and analyze flaws in browser security policies. Previous work has looked at weaknesses in cross-frame communication mechanisms [32], frame navigation policies [32,129], client-side browser state [78], cookie path protection [100], protection among documents within same origin [75], display protection [129], and other issues. Zalewski [138] documents the security design in browsers including some loopholes. This proposed research complements these efforts by providing a more systematic approach for identifying incoherencies in browser's access control policies. To our knowledge, this would be the first, principal-driven analysis on browsers' access control policies.

DOM access checker [139] is a tool designed to automatically validate numerous aspects of domain security policy enforcement (cross-domain DOM access, JavaScript cookies, XMLHttpRequest calls, event and transition handling) to detect common security attack or information disclosure vectors. Browserscope [4] is a community-driven project for tracking browser functionality. Its security test suite [74] checks whether new browser security features are implemented by a browser. In our analysis of access control incoherency, we will focus on uncovering the incoherencies from examining the interplay between resources, runtime identity changes, and the user principal's resource access control. This focus and methodology differ from these previous or ongoing work and our analysis not only touches on DOM, but also on the HTTP network layer and display.

6.3.1 Web Evaluation Frameworks

Compared to previous work, a unique aspect of our proposed research will be our extensive evaluation of the cost of removing unsafe policies from the current web by actively crawling and executing web content. Yue et al. [137] also used a crawling-based, execution-based approach to measure the prevalence of unsafe JavaScript features on 6,805 popular web sites. They used a JavaScript interposition technique that is similar to our script interposition, but they lack any network and display interposition capabilities, limiting the policies they can monitor. As well, we will use a significantly larger dataset.

Our active crawling infrastructure will build on previous efforts that have analyzed safety of web pages by rendering them in real browsers running within virtual machines [95,96,105,106,131]. We will extend these frameworks with additional browser interposition support to monitor unsafe browser security policies.

6.4 Content Integrity

Prior work has explored a number of integrity protection techniques. A proposal on authentication-only ciphersuites for PSK-TLS [41] describes a transport layer security scheme for authentication and integrity, with no confidentiality guarantees. However, this proposal requires a shared secret between each client and the server to key the hash, making it impractical to share the key with all the clients of the application. Our work builds on SHTTP [110]'s signature mode of operation and additionally addressed progressive content loading and the associated security.

Web tripwires [108] verify the integrity of a page by matching it against a known good representation of the page (either a checksum or an encoded full copy of the page's HTML). It uses client-side JavaScript code to detect in-flight modifications to a web page. However, web tripwires have high network overhead (approximately 17% of the page size), which could hinder the end-to-end response time, especially for slower networks. Moreover, web tripwires can be identified and disabled by an adversary, and they cannot detect full-page substitutions. In contrast, HTTPi is cryptographically secure and can prevent any type of integrity breaches. HTTPi also has a much lower network overhead cost as compared to web tripwire. Finally, web tripwires focus on *detection*, while HTTPi focuses on both *detection* and *prevention*.

HTTP provides a Content-MD5 header [58] that can carry the MD5 signature of the complete page. This header could be useful in providing basic page integrity, but suffers from many weaknesses if used by itself. For example, a network attacker can modify the header since it is not authenticated, and the attacker can completely drop the header without the client knowing about it. In contrast, HTTPi provides authentication by signing content hashes, and since it specifies the requirements for a page using HSTS in advance, the client can easily detect whether the required integrity content is dropped by network attackers. Additionally, with HTTPi, integrity is

evaluated over smaller-sized segments, which has performance benefits over the entire-page approach used in the Content-MD5 header.

The YURL [47] specification defines an alternative server identification and authentication mechanism that does not depend on centralized authorities like the DNS or PKI. A YURL identifies a site using the site's public key fingerprint and the web site owner owns the CA fingerprint. However, like HTTPS, and unlike HTTPi, the proposed YURL-based protocol *httpsy* [47] precludes content from being cached at web proxies.

CHAPTER VII

CONCLUSIONS

7.1 Summary

The Web continues to evolve creating new challenges and requirements for systems that are currently deployed on the Web. The introduction of new web players, such as the average users who contribute web content, further impose new requirements. The available security policies and protection mechanisms continue to be stretched exposing their limitations and potential vulnerabilities. In this dissertation, we have addressed this important issue and evaluated the current security and privacy policies in view of the changing Web requirements. We have also developed enforcement frameworks that effectively ensure that the security policies specified by different web players are correctly followed. We have addressed several challenges in view of our goal to improve end-to-end security for web access and made four unique contributions pertaining to policy specification and enforcement.

First, we developed xAccess, an application-independent, generalized framework that allows average users to define access control policies for their contributed data. The contributions associated with this piece of work are as follows:

- We provided a novel design of an unified access control framework for supporting diverse user-defined access control policies that empowers the users to choose their own models and their own access granularity. We also show that our model is *generic* to allow simulation of a number of popular access control models on top of our framework, and provides enormous *flexibility* to the users in making access control decisions about the data owned by them.

- We developed a proof-of-concept prototype system for xAccess that provides a

set of APIs that can be used to integrate generalized access control capability into web applications.

- We demonstrated the viability of our framework by developing a sample blogging application as our base example and subsequently integrating access control into the application using the xAccess APIs. We also showed real-world deployment potential of our framework by integrating xAccess into a popular open-source wikipedia application.

The next part of the thesis focused on extending the enforcement of user-defined policies to third-party applications. We developed xBook, a framework that uses information flow control models to ensure that untrusted third-party applications only share data within the realms of the user's privacy policies, thus preventing any accidental or malicious leaks by these applications. This work made multiple contributions:

- We presented a novel design of a social networking platform for supporting third-party applications that not only controls what user profile data the applications are allowed to access, but also mediates what the applications can do with the user data they can access.

- We developed a proof-of-concept system for xBook that provides a set of APIs that can be used to develop privacy-preserving social networking applications.

- We showed the viability of our framework by porting the functionality of some popular Facebook applications to applications developed using the xBook APIs.

- We demonstrated a practical deployment strategy of our system by porting our platform itself as an application on Facebook.

For the third component of the thesis, the web player of focus was the client-side software, specifically the web browser. We studied the current state of access control

policies that browsers use to share resources among their web site principals. We showed that mishandling of such principals leads to many access control incoherencies, presenting hurdles for web developers to construct secure web applications. In summary, this work made the following contributions:

- We provided a systematic, principal-driven analysis of access control incoherencies in today's browsers.
- We introduced the user principal concept for the browser setting.
- We developed a comprehensive, extensible compatibility measurement framework to measure the use of various browser features on the Web.
- We performed the first large-scale measurements on the compatibility cost of coherent access control policies. Our methodology and results serve as a guideline for browser designers to balance security and backward compatibility.

Finally, this dissertation presented a mechanism that enables web applications to define policies to balance their desired security and performance. We observed that only end-to-end authentication and integrity are required for the browser platform to enforce its access control reliably. Without end-to-end confidentiality, content can be cached. Subsequently, we proposed a novel protocol, called HTTPi, which offers only end-to-end authentication and integrity and seamlessly works with the existing web caching infrastructure. This allows web applications to cache their content in the network while ensuring that the content's integrity is protected. The contributions of this work are:

- We presented an end-to-end design and implementation of HTTPi, that does not require changes at intermediate nodes, such as proxies, for caching HTTPi content. Our design also ensures that progressive content loading in browsers is not hindered by HTTPi, and that this incurs minimal overhead in both computation and bandwidth.

- We performed web measurements to show that a significant portion of existing HTTPS content can be shared and cached across users. These measurements demonstrate the importance of HTTPi as they indicate that much of existing HTTPS content can be safely turned into HTTPi content to achieve better performance and the ability to offload content to other servers without any loss of security.

- We developed a mechanism that provided web developers with an easy way to specify policies on how the HTTPi content can be safely mixed together with HTTP and HTTPS content. Furthermore, we observed that the default isolation policy for HTTPi, HTTPS, and HTTP content of the same domain and port does not need to be as strict as the same-origin policy. To this end, we designed a new default policy to allow useful interactions across different protocol schemes without sacrificing security.

7.2 Future Work

While a number of problems regarding the design and enforcement of web security policies have been addressed in this thesis, the research work has revealed several other areas to explore and a few open problems that may be worth solving in order to reach the goal of improving the effectiveness of web security solutions. These problems and areas are described below:

- **Web security for mobile platforms.** Mobile devices are limited in their capability in processing, storage or screen space, and this limitation creates additional challenges to accessing the Web securely on these devices. On one hand, the security design of the mobile applications rendering the Web on these mobile devices is limited; on the other hand, the mobile version of the web applications create security pitfalls that can be exploited. The mobility of these devices also brings additional security challenges. Our work on analyzing

policies in the desktop space can be extended to analyze the effectiveness of the web security policies in the mobile environment [28] and develop robust, secure system designs that are more aligned with the requirements of the mobile web. The associated challenges in the mobile web space need to be further investigated and explored.

- **Cloud-based web application designs.** Cloud computing has fast emerged as a cornerstone technology for enterprises to offload bulk of their processing and storage needs to the server clouds. This trend has had a noticeable impact on how users access their applications (such as documents or email): instead of the users being individually responsible for deploying these applications, the applications are shifting base to the cloud providers who are now responsible for the deployment of these applications. This has also resulted in the security of these applications being shifted from the users to the cloud providers, while users' demands for security and data privacy remain unchanged. On one hand, the cloud providers are hosting large amount of sensitive user data; on the other hand, they are hosting mutually distrusting applications that seek access of user data to perform their functionality. This makes the cloud environment an attractive target for malicious attackers.

  The increasing growth of capability-limited mobile devices mixed with a strategic shift towards cloud computing leads to many interesting design possibilities in the web space. The cloud is moving towards a programming platform model providing interfaces for applications to utilize the cloud services. The cloud provider would need to enforce users' security policies on the data contributed by the users. At the same time, mandatory security policies need to be developed and analyzed for the cloud environment.

- **User-centric security framework designs.** In this thesis, we explored a

design of a generalized access control frameworks that allow users' to enforce their security and privacy policies independent of the underlying application. We believe that the transition of the cloud to become service providers would enable such designs. However, certain applications, such as healthcare applications, would impose additional security requirements due to the sensitivity of the information involved and the players involved in deciding the security policies (such as government laws, hospital policies, etc.). Further work is needed to examine how application-specific requirements impact the design of a generalized user-centric secure systems and how such systems can be customized with minimum design changes and deployment hurdles.

- **Semantic Web.** The next generation of web would involve associating meaning and description to the web content, a concept often referred to as the semantic Web [21]. The semantic Web is a vision of information that can be interpreted by automated processes, so these processes can perform more of the tedious work involved in finding, combining, and acting upon information on the web and presenting it in a user-friendly form to the end user. This would create new challenges in providing privacy protection to users' personal data and subsequently would require design changes to the mechanisms for enforcing users' security and privacy policies.

7.3 Closing Remarks

This thesis addressed several research problems with the goal to bridge the gap between the evolving Web 2.0 environment and the available security policies and mechanisms that determine the end-to-end security of web content. We analyzed the effectiveness of the current security and privacy policies in view of the new challenges imposed by the dynamic Web. We also developed flexible designs of new security enforcement frameworks that overcome the limitations of the current web designs in

providing better security guarantees and effective policy enforcement. Research on the open problems along this line can improve the effectiveness of end-to-end security mechanisms on the Web. Looking ahead, similar challenges need to be addressed in emerging areas of mobile and cloud computing. The Web would continue to evolve with new additional features and mechanisms. A more organized and systematic effort is needed to design security policies for such new features in order for them to be effective and coherent.

REFERENCES

[1] "About Asynchronous Pluggable Protocols." http://msdn.microsoft.com/en-us/library/aa767916(v=VS.85).aspx. Accessed on Sept. 12, 2010.

[2] "ADsafe." http://adsafe.org. Accessed on Feb. 1, 2009.

[3] "Amazon Elastic Computing Cloud." http://aws.amazon.com/ec2/. Accessed on Feb. 1, 2009.

[4] "Browserscope." http://www.browserscope.org/.

[5] "BugMeNot." http://www.bugmenot.com/. Last accessed Mar. 1, 2010.

[6] "Cisco Visual Networking Index: Forecast and Methodology, 2009-2014," http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html. Accessed on Sept. 23, 2010.

[7] "Daily Horoscopes." http://apps.facebook.com/daily-horoscope. Accessed on Feb. 1, 2009.

[8] "Dispelling the New SSL Myth." http://devcentral.f5.com/weblogs/macvittie/archive/2011/01/31/dispelling-the-new-ssl-myth.aspx. Accessed on May 1, 2011.

[9] "Facebook Developers: Developer Terms of Service." http://developers.facebook.com/terms.php. Accessed on Feb. 1, 2009.

[10] "Facebook's Privacy Policy." http://www.facebook.com/policy.php. Accessed on Feb. 1, 2009.

[11] "Helma Javascript Web Application Framework." http://www.helma.org. Accessed on Feb. 1, 2009.

[12] "IInternetProtocolInfo interface." http://msdn.microsoft.com/en-us/library/aa767874(VS.85).aspx. Accessed on Sept. 12, 2010.

[13] "JavaScript Object Notation (JSON)." http://www.json.org. Accessed on Feb. 1, 2009.

[14] "JSLint: The JavaScript Verifier." http://www.jslint.com. Accessed on Feb. 1, 2009.

[15] "Map Your Friends." http://apps.facebook.com/mapyourfriends. Accessed on Feb. 1, 2009.

[16] "mediaWiki with xAccess." http://www.own-provide-decide.com:8001/mediawiki.

[17] "Network Simulator." http://www.akmalabs.com/downloads_netsim.php. Accessed on Sept. 12, 2010.

[18] "OpenID." http://openid.net.

[19] "OpenSocial." http://www.opensocial.org/. Accessed on Feb. 1, 2009.

[20] "Orkut." http://www.orkut.com.

[21] "Semantic Web." http://semanticweb.org/. Accessed on May 1, 2011.

[22] "TopFriends." http://apps.facebook.com/topfriends. Accessed on Feb. 1, 2009.

[23] "xBlog with xAccess." http://www.own-provide-decide.com/accesscontrol/static/CLIENT/login.html.

[24] "What's New in Internet Explorer 8," 2008. http://msdn.microsoft.com/en-us/library/cc288472.aspx. Accessed on Nov. 14, 2009.

[25] Acharya, A. and Raje, M., "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium*, (Denver, CO), Aug. 2000.

[26] Akamai Technologies, I., "Secure Content Delivery." http://www.akamai.com/dl/feature_sheets/fs_edgesuite_securecontentdelivery.pdf.

[27] "Alexa." http://www.alexa.com/.

[28] Amrutkar, C., Singh, K., Verma, A., and Traynor, P., "On the Disparity of Display Security in Mobile and Traditional Web Browsers," Tech. Rep. GT-CS-11-02, Georgia Institute of Technology, Atlanta, GA, Jan. 2011.

[29] Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S., "RFC 4033: DNS Security Introduction and Requirements," 2005.

[30] Backstrom, L., Dwork, C., and Kleinberg, J., "Wherefore Art Thou R3579X?: Anonymized Social Networks, Hidden Patterns, and Structural Steganography," in *Proceedings of the 16th International Conference on World Wide Web (WWW)*, (Banff, Canada), May 2007.

[31] Barth, A., "HTTP State Management Mechanism." IETF Draft 2109, Feb 2010. http://tools.ietf.org/html/draft-ietf-httpstate-cookie-03.

[32] Barth, A., Jackson, C., and Mitchell, J. C., "Securing Frame Communication in Browsers," in *Proceedings of the 17th USENIX Security Symposium*, (San Jose, CA), July 2008.

[33] Bell, D. E. and LaPadula, L. J., "Secure Computer Systems: Mathematical Foundations," Tech. Rep. ESD-TR-73-278, MITRE Corporation, Bedford, MA, Nov. 1973.

[34] Bell, D. E. and Lapadula, L. J., "Secure Computer System: Unified Exposition and Multics Interpretation," Tech. Rep. MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976.

[35] Bell, D. E., "Looking Back at the Bell-LaPadula Model," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, (Tucson, AZ), Dec. 2005.

[36] Benantar, M., *Access Control Systems: Security, Identity Management and Trust Models*. Secaucus, NJ: Springer-Verlag, 2005.

[37] Benslimane, D., Dustdar, S., and Sheth, A., "Services Mashups: The New Generation of Web Applications," *IEEE Internet Computing*, vol. 12, pp. 13–15, Sept. 2008.

[38] Bertino, E., Fan, J., Ferrari, E., Hacid, M.-S., Elmagarmid, A. K., and Zhu, X., "A Hierarchical Access Control Model for Video Database Systems," *ACM Transactions on Information Systems*, vol. 21, no. 2, pp. 155–191, 2003.

[39] Biba, K. J., "Integrity Considerations for Secure Computer Systems," Tech. Rep. ESD-TR-76-372, MITRE Corporation, Bedford, MA, Apr. 1977.

[40] Blum, A., Dwork, C., McSherry, F., and Nissim, K., "Practical Privacy: the SuLQ Framework," in *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, (Baltimore, MD), 2005.

[41] Blumenthal, U. and Goel, P., "RFC 4785: Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)," 2007.

[42] Byun, J.-W., Bertino, E., and Li, N., "Purpose-Based Access Control of Complex Data for Privacy Protection," in *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT)*, (Stockholm, Sweden), June 2005.

[43] Carminati, B., Ferrari, E., and Perego, A., "Enforcing Access Control in Web-based Social Networks," *ACM Transactions on Information and Systems Security*, 2008.

[44] Chinaei, A. H. and Tompa, F. W., "User-Managed Access Control for Health Care Systems," in *Secure Data Management Workshop*, (Trondheim, Norway), Sept. 2005.

[45] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X., "Secure Web Applications via Automatic Partitioning," in *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, (Stevenson, WA), Oct. 2007.

[46] "Clickjacking." `http://en.wikipedia.org/wiki/Clickjacking`.

[47] Close, T., "Petname Tool: Enabling Web site Recognition using the Existing SSL Infrastructure," in *W3C Workshop on Transparency and Usability of Web Authentication*, (New York, NY), Mar. 2006.

[48] Crane, D., Pascarello, E., and James, D., *Ajax in Action*. Greenwich, CT, USA: Manning Publications Co., 2005.

[49] Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P., "A Fine-grained Access Control System for XML Documents," *ACM Transactions on Information Systems Security*, vol. 5, no. 2, pp. 169–202, 2002.

[50] DeGrande, K., "CDNetworks," September 2010. Personal communication.

[51] Denning, D. E., "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[52] "Document Object Model." `http://www.w3.org/DOM/`. Accessed on Nov. 14, 2009.

[53] Eckersley, P. and Burns, J., "Observatory for the SSLiverse," July 2010. `http://www.eff.org/files/DefconSSLiverse.pdf`.

[54] Enck, W., Rueda, S., Schiffman, J., Sreenivasan, Y., Clair, L. S., Jaeger, T., and McDaniel, P., "Protecting users from "themselves"," in *ACM Workshop on Computer Security Architecture*, (Fairfax, VA), Nov. 2007.

[55] Felt, A. and Evans, D., "Privacy Protection for Social Networking Platforms," in *Web 2.0 Security and Privacy Workshop (W2SP)*, (Oakland, CA), May 2008.

[56] Ferraiolo, D. F., Cuigini, J. A., and Kuhn, D. R., "Role-Based Access Control (RBAC): Features and Motivations," in *Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC)*, (New Orleans, LA), dec 1995.

[57] "FiddlerCore." `http://fiddler.wikidot.com/fiddlercore`. Accessed on Nov. 14, 2009.

[58] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "RFC2616: Hypertext Transfer Protocol – HTTP/1.1," 1999.

[59] Finifter, M., Mettler, A., Sastry, N., and Wagner, D., "Verifiable Functional Purity in Java," in *Proceedings of the 15th ACM Conference on Computer and Communication Security (CCS)*, (Alexandria, VA), Oct. 2008.

[60] Flanagan, D., *Javascript: The Definitive Guide*. O'Reilly Media Inc., 2006.

[61] Friedl, S., "An Illustrated Guide to the Kaminsky DNS Vulnerability." `http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html`.

[62] Garrett, J. J., "Ajax: A New Approach to Web Applications," Feb. 2005. `http://www.adaptivepath.com/ideas/e000385`. Accessed on May 1, 2011.

[63] Gaspard, C., Goldberg, S., Itani, W., Bertino, E., and Nita-Rotaru, C., "SINE: Cache-Friendly Integrity for the Web," in *Workshop on Secure Network Protocols (NPSec)*, (Princeton, NJ), Oct. 2009.

[64] Gates, C., "Access Control Requirements for Web 2.0 Security and Privacy," in *Web 2.0 Security and Privacy Workshop (W2SP)*, (Oakland, CA), May 2007.

[65] Gennaro, R. and Rohatgi, P., "How to Sign Digital Streams," in *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, (Santa Barbara, CA), Aug. 1997.

[66] Graham, P., "Web 2.0," Nov. 2005. `http://www.paulgraham.com/web20.html`. Accessed on May 1, 2011.

[67] Hacking, S., "More Advertising Issues on Facebook (Updated)," June 2008. `http://theharmonyguy.com/2008/06/20/more-advertising-issues-on-facebook/`. Accessed on Feb. 1, 2009.

[68] Hanna, S., Shin, R., Akhawe, D., Saxena, P., Boehm, A., and Song, D., "The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives," in *Web 2.0 Security and Privacy Worskhop (W2SP)*, (Oakland, CA), May 2010.

[69] Hart, M., Johnson, R., and Stent, A., "More Content – Less Control: Access Control in the Web 2.0," in *Web 2.0 Security and Privacy Workshop (W2SP)*, (Oakland, CA), May 2008.

[70] He, Y.-Z., Han, Z., and Du, Y., "Configuring RBAC to Simulate Bell Model," in *International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, (Harbin, China), Aug. 2008.

[71] Hodges, J., Jackson, C., and Barth, A., "HTTP Strict Transport Security (HSTS)," 2010. `http://tools.ietf.org/html/draft-hodges-strict-transport-sec`.

[72] "HTML 5 Editor's Draft," October 2008. `http://www.w3.org/html/wg/html5/`.

[73] "HttpOnly." http://www.owasp.org/index.php/HTTPOnly. Accessed on Nov. 14, 2009.

[74] Jackson, C. and Barth, A., "Browserscope Security Test Suite." http://mayscript.com/blog/collinj/browserscope-security-test-suite. Accessed on Nov. 14, 2009.

[75] Jackson, C. and Barth, A., "Beware of Finer-Grained Origins," in *Web 2.0 Security and Privacy Workshop (W2SP)*, (Oakland, CA), May 2008.

[76] Jackson, C. and Barth, A., "ForceHTTPS: Protecting High-Security Web Sites from Network Attacks," in *Proceedings of the 17th International Conference on World Wide Web (WWW)*, (Beijing, China), Apr. 2008.

[77] Jackson, C., Barth, A., Bortz, A., Shao, W., and Boneh, D., "Protecting Browsers from DNS Rebinding Attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), Oct. 2007.

[78] Jackson, C., Bortz, A., Boneh, D., and Mitchell, J. C., "Protecting Browser State from Web Privacy Attacks," in *Proceedings of the 15th International Conference on World Wide Web (WWW)*, (Edinburgh, Scotland), May 2006.

[79] Jirasek, V., "Overcoming man in the middle attack on Strict Transport Security," August 2010. http://blog.jirasek.eu/2010/08/overcoming-man-in-middle-attack-on.html.

[80] Konrad, R., "Facebook opens to third-party developers," May 2007. http://www.msnbc.msn.com/id/18899269/. Accessed on Feb. 1, 2009.

[81] Kristol, D. and Montulli, L., "HTTP State Management Mechanism," in *IETF RFC 2965*, Oct. 2000.

[82] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R., "Information Flow Control for Standard OS Abstractions," in *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP)*, (Stevenson, WA), Oct. 2007.

[83] Langley, A., Modadugu, N., and Chang, W.-T., "Overclocking SSL," in *Velocity: Web Performance and Operations Conference*, (Santa Clara, CA), June 2010. http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html. Accessed on Sept. 12, 2010.

[84] Lawrence, E., "Fiddler Web Debugging Tool." http://www.fiddler2.com/fiddler2/. Accessed on Nov. 14, 2009.

[85] Lenhart, A. and Fox, S., "Bloggers: A Potrait of the Internet's New Storytellers," July 2006. http://www.pewinternet.org/~/media//Files/Reports/2006/PIPBloggersReportJuly192006.pdf.pdf.

[86] Li, N. and Mao, Z., "Administration in role-based access control," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, (Singapore), Mar. 2007.

[87] Li, N. and Tripunitara, M. V., "On Safety in Discretionary Access Control," in *Proceedings of the 26th IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2005.

[88] Li, N., Tripunitara, M. V., and Wang, Q., "Resiliency Policies in Access Control," in *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS)*, (Alexandria, VA), Oct. 2006.

[89] Machanavajjhala, A., Kifer, D., Gehrke, J., and Venkitasubramaniam, M., "L-diversity: Privacy beyond k-anonymity," *ACM Transactions of Knowledge Discovery from Data*, vol. 1, no. 1, p. 3, 2007.

[90] Mecella, M., Ouzzani, M., Paci, F., and Bertino, E., "Access Control Enforcement for Conversation-based Web Services," in *Proceedings of the 15th International Conference on World Wide Web (WWW)*, (Edinburgh, Scotland), May 2006.

[91] Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M., "Caja: Safe Active Content in Sanitized JavaScript," Oct. 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[92] Mills, E., "Facebook suspends app that permitted peephole," June 2008. http://news.cnet.com/8301-10784_3-9977762-7.html. Accessed on Feb. 1, 2009.

[93] "Mitigating Cross-site Scripting With HTTP-only Cookies." http://msdn2.microsoft.com/en-us/library/ms533046.aspx. Accessed on Nov. 14, 2009.

[94] Mosberger, D. and Jin, T., "httperf—A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[95] Moshchuk, A., Bragin, T., Deville, D., Gribble, S. D., and Levy, H. M., "SpyProxy: Execution-based Detection of Malicious Web Content," in *Proceedings of the 16th USENIX Security Symposium*, (Boston, MA), Aug. 2007.

[96] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M., "A Crawler-based Study of Spyware on the Web," in *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS)*, (San Diego, CA), Feb. 2006.

[97] "Mozilla Foundation Security Advisory 2009-05: XMLHttpRequest allows reading HTTPOnly cookies." http://www.mozilla.org/security/announce/2009/mfsa2009-05.html. Accessed on Nov. 14, 2009.

[98] Myers, A. C. and Liskov, B., "A Decentralized Model for Information Flow Control," in *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)*, (Saint-Malo, France), Oct. 1997.

[99] Narayanan, A. and Shmatikov, V., "Robust De-anonymization of Large Sparse Datasets," in *Proceedings of the 29th IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2008.

[100] O'Neal, M., "Cookie Path Best Practice." http://research.corsaire.com/whitepapers/040323-cookie-path-best-practice.pdf.

[101] O'Reilly, T., "What Is Web 2.0," Sept. 2005. http://oreilly.com/web2/archive/what-is-web-20.html. Accessed on May 1, 2011.

[102] Osborn, S., Sandhu, R., and Munawer, Q., "Configuring Role-based Access Control to Enforce Mandatory and Discretionary Access Control Policies," *ACM Transactions on Information and Systems Security*, vol. 3, no. 2, pp. 85–106, 2000.

[103] Panja, T., "Oxford using Facebook to snoop." http://www.msnbc.msn.com/id/19813092/. Accessed on Feb. 1, 2009.

[104] Peterson, D. S., Bishop, M., and Pandey, R., "A Flexible Containment Mechanism for Executing Untrusted Code," in *Proceedings of the 11th USENIX Security Symposium*, (San Franscisco, CA), Aug. 2002.

[105] Provos, N., Mavrommatis, P., Rajab, M., and Monrose, F., "All Your iFrames Point to Us," in *Proceedings of the 17th USENIX Security Symposium*, (San Jose, CA), July 2008.

[106] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N., "The ghost in the browser: analysis of web-based malware," in *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets (HotBots)*, (Cambridge, MA), Apr. 2007.

[107] Ramachandran, S., "Let's make the web faster." http://code.google.com/speed/articles/web-metrics.html. Accessed on Sept. 12, 2010.

[108] Reis, C., Gribble, S. D., Kohno, T., and Weaver, N. C., "Detecting In-Flight Page Changes with Web Tripwires," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (San Francisco, CA), Apr. 2008.

[109] Rescorla, E., "RFC 2818: HTTP Over TLS," 2000.

[110] Rescorla., E. and Schiffman, A., "RFC2660: The Secure Hypertext Transfer Protocol," 1999.

[111] Ruderman, J., "Same Origin Policy for JavaScript." `http://www.mozilla.org/projects/security/components/same-origin.html`. Accessed on Nov. 14, 2009.

[112] Samarati, P., "Protecting Respondents' Identities in Microdata Release," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 6, pp. 1010–1027, 2001.

[113] Samarati, P., Bertino, E., and Jajodia, S., "An Authorization Model for a Distributed Hypertext System," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 4, pp. 555–562, 1996.

[114] Sandhu, R. and Munawer, Q., "How to do Discretionary Access Control using Roles," in *ACM Workshop on Role-Based Access Control*, (Fairfax, VA), Oct. 1998.

[115] Sandhu, R. S., "Lattice-Based Access Control Models," *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[116] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E., "Role-Based Access Control Models," *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[117] Schechter, S., Dhamija, R., Ozment, A., and Fischer, I., "The Emperor's New Security Indicators," in *Proceedings of the 28th IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2007.

[118] Sciba, D., "Mayor in MySpace photo flap asked to resign." `http://www.katu.com/news/13670287.html`. Accessed on Feb. 1, 2009.

[119] Shneiderman, B., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3rd ed., 1998.

[120] Simpson, A., "On the Need for User-defined Fine-grained Access Control Policies for Social Networking Applications," in *Workshop on Security in Opportunistic and SOCial networks*, (Istanbul, Turkey), Sept. 2008.

[121] Singh, K., Bhola, S., and Lee, W., "xBook: Redesigning Privacy Control in Social Networking Platforms," in *Proceedings of the 18th USENIX Security Symposium*, (Montreal, Canada), Aug. 2009.

[122] Singh, K., Erete, I., and Lee, W., "I Own, I Provide, I Decide: Generalized User-Centric Access Control Framework for Web Applications," Tech. Rep. GT-CS-10-22, Georgia Institute of Technology, Atlanta, GA, Dec. 2010.

[123] Singh, K., Moshchuk, A., Wang, H. J., and Lee, W., "On the Incoherencies in Web Browser Access Control Policies," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2010.

[124] Singh, K., Wang, H. J., Moshchuk, A., Jackson, C., and Lee, W., "HTTPi for Practical End-to-End Web Content Integrity," Tech. Rep. MSR-TR-2011-63, Microsoft Research, Redmond, WA, May 2011.

[125] Stamm, S., Sterne, B., and Markham, G., "Reining in the Web with Content Security Policy," in *Proceedings of the 19th International World Wide Web Conference (WWW)*, (Raleigh, NC), Apr. 2010.

[126] Sweeney, L., "Weaving Technology and Policy Together to Maintain Confidentiality," *Journal of Law, Medicine and Ethics*, vol. 25, pp. 98–110, 1997.

[127] Tootoonchian, A., Gollu, K. K., Saroiu, S., Ganjali, Y., and Wolman, A., "Lockr: Social Access Control for Web 2.0," in *Workshop on Online Social Networks*, (Seattle, WA), Aug. 2008.

[128] Wang, H. J., Fan, X., Howell, J., and Jackson, C., "Protection and Communication Abstractions for Web Browsers in MashupOS," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, (Stevenson, WA), Oct. 2007.

[129] Wang, H. J., Grier, C., Moshchuk, A., King, S. T., Choudhury, P., and Venter, H., "The Multi-Principal OS Construction of the Gazelle Web Browser," in *Proceedings of the 18th USENIX Security Symposium*, (Montreal, Canada), Aug. 2009.

[130] Wang, J., "A Survey of Web Caching Schemes for the Internet," *SIGCOMM Computer Communication Review*, vol. 29, pp. 36–46, October 1999.

[131] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S., "Automated Web Patrol with Strider HoneyMonkeys," in *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS)*, (San Diego, CA), Feb. 2006.

[132] Williams, C., "Facebook application hawks your personal opinions for cash," Sept. 2007. `http://www.theregister.co.uk/2007/09/12/facebook_compare_people/`. Accessed on Feb. 1, 2009.

[133] Wolman, A., Voelker, G. M., Sharma, N., Cardwell, N., Karlin, A., and Levy, H. M., "On the Scale and Performance of Cooperative Web Proxy Caching," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, (Charleston, SC), Dec. 1999.

[134] "XMLHttpRequest." `http://www.w3.org/TR/XMLHttpRequest/`. Accessed on Nov. 14, 2009.

[135] "XMLHttpRequest Level 2." http://www.w3.org/TR/XMLHttpRequest2/. Accessed on Aug. 10, 2009.

[136] Yu, J., Benatallah, B., Casati, F., and Daniel, F., "Understanding Mashup Development," *IEEE Internet Computing*, vol. 12, pp. 44–52, Sept. 2008.

[137] Yue, C. and Wang, H., "Characterizing Insecure JavaScript Practices on the Web," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, (Madrid, Spain), Apr. 2009.

[138] Zalewski, M., "Browser Security Handbook," 2008. http://code.google.com/p/browsersec/wiki/Main.

[139] Zalewski, M. and Almeida, F., "Browser DOM Access Checker 1.01." http://lcamtuf.coredump.cx/dom_checker/. Accessed on Nov. 14, 2009.

[140] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D., "Making Information Flow Explicit in HiStar," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, (Seattle, WA), Nov. 2006.

[141] Zhang, Z., Zhang, X., and Sandhu, R., "ROBAC: Scalable Role and Organization Based Access Control Models," in *Proceedings of the 2nd International Conference on Collaborative Computing (CollaborateCom)*, (Atlanta, GA), Nov. 2006.